



Intel® Math Kernel Library

Developer Reference

Revision: 076

MKL 11.3

Legal Information

Contents

Legal Information	31
Introducing the Intel® Math Kernel Library	37
Getting Help and Support	39
What's New	41
Notational Conventions	43
Chapter 1: Function Domains	
Performance Enhancements.....	48
Parallelism.....	49
Chapter 2: BLAS and Sparse BLAS Routines	
BLAS Routines.....	51
Routine Naming Conventions.....	51
Fortran 95 Interface Conventions.....	53
Matrix Storage Schemes.....	53
BLAS Level 1 Routines and Functions.....	54
?asum.....	54
?axpy.....	56
?copy.....	57
?dot.....	58
?sdot.....	59
?dotc.....	60
?dotu.....	61
?nrm2.....	63
?rot.....	64
?rotg.....	65
?rotm.....	66
?rotmg.....	68
?scal.....	69
?swap.....	71
i?amax.....	72
i?amin.....	73
?cabs1.....	74
BLAS Level 2 Routines.....	74
?gbmv.....	75
?gemv.....	78
?ger.....	81
?gerc.....	82
?geru.....	84
?hbmh.....	85
?hemv.....	88
?her.....	90
?her2.....	92
?hpmv.....	93
?hpr.....	95

?hpr2.....	97
?sbmv.....	99
?spmv.....	101
?spr.....	103
?spr2.....	105
?symv.....	107
?syr.....	109
?syr2.....	110
?tbmv.....	112
?tbsv.....	114
?tpmv.....	117
?tpsv.....	119
?trmv.....	121
?trsv.....	123
BLAS Level 3 Routines.....	125
?gemm.....	126
?hemm.....	129
?gemm_batch.....	131
?herk.....	135
?her2k.....	137
?symm.....	140
?syrk.....	143
?syr2k.....	145
?trmm.....	148
?trsm.....	151
Sparse BLAS Level 1 Routines.....	153
Vector Arguments.....	154
Naming Conventions.....	154
Routines and Data Types.....	154
BLAS Level 1 Routines That Can Work With Sparse Vectors.....	155
?axpyi.....	155
?doti.....	157
?dotci.....	158
?dotui.....	159
?gthr.....	160
?gthrz.....	161
?roti.....	162
?sctr.....	164
Sparse BLAS Level 2 and Level 3 Routines.....	165
Naming Conventions in Sparse BLAS Level 2 and Level 3.....	165
Sparse Matrix Storage Formats.....	166
Routines and Supported Operations.....	167
Interface Consideration.....	168
Sparse BLAS Level 2 and Level 3 Routines.....	172
mkl_?csrgev.....	175
mkl_?bsrgev.....	178
mkl_?coogev.....	180
mkl_?diagev.....	182
mkl_?csrsvmv.....	184
mkl_?bsrsvmv.....	187

mkl_?coosymv.....	189
mkl_?diasymv.....	191
mkl_?csrtrsv.....	193
mkl_?bsrtrsv.....	196
mkl_?cootrsv.....	199
mkl_?diatrsv.....	202
mkl_cspblas_?csrgemv.....	204
mkl_cspblas_?bsrgemv.....	207
mkl_cspblas_?coogemv.....	209
mkl_cspblas_?csrsymv.....	211
mkl_cspblas_?bsrsymv.....	214
mkl_cspblas_?coosymv.....	216
mkl_cspblas_?csrtrsv.....	218
mkl_cspblas_?bsrtrsv.....	221
mkl_cspblas_?cootrsv.....	224
mkl_?csrmm.....	226
mkl_?bsrmm.....	230
mkl_?cscmv.....	233
mkl_?coomv.....	237
mkl_?csrsv.....	240
mkl_?bsrsv.....	243
mkl_?cscsv.....	247
mkl_?coosv.....	250
mkl_?csrmm.....	253
mkl_?bsrmm.....	257
mkl_?cscmm.....	261
mkl_?coomm.....	265
mkl_?csrsm.....	268
mkl_?cscsm.....	272
mkl_?coosm.....	276
mkl_?bsrsm.....	279
mkl_?diamv.....	282
mkl_?skymv.....	286
mkl_?diasv.....	289
mkl_?skysv.....	292
mkl_?diamm.....	295
mkl_?skymm.....	298
mkl_?diasm.....	302
mkl_?skysm.....	305
mkl_?dnscsr.....	308
mkl_?csrcoo.....	311
mkl_?csrbsr.....	314
mkl_?csrsc.....	317
mkl_?csrdia.....	320
mkl_?csrsky.....	323
mkl_?csradd.....	326
mkl_?csrmultcsr.....	330
mkl_?csrmultd.....	334
Inspector-executor Sparse BLAS Routines.....	337
Naming conventions in Inspector-executor Sparse BLAS Routines.....	337

Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS	
Routines.....	338
Supported Inspector-executor Sparse BLAS Operations.....	338
Matrix manipulation routines.....	339
mkl_sparse_?_create_csr.....	340
mkl_sparse_?_create_csc.....	342
mkl_sparse_?_create_coo.....	343
mkl_sparse_?_create_bsr.....	345
mkl_sparse_copy.....	347
mkl_sparse_destroy.....	349
mkl_sparse_convert_csr.....	350
mkl_sparse_convert_bsr.....	351
mkl_sparse_?_export_csr.....	352
mkl_sparse_?_export_bsr.....	354
mkl_sparse_?_set_value.....	357
Inspector-executor Sparse BLAS Analysis Routines.....	358
mkl_sparse_set_mv_hint.....	358
mkl_sparse_set_sv_hint.....	360
mkl_sparse_set_mm_hint.....	363
mkl_sparse_set_sm_hint.....	365
mkl_sparse_set_memory_hint.....	367
mkl_sparse_optimize.....	369
Inspector-executor Sparse BLAS Execution Routines.....	370
mkl_sparse_?_mv.....	370
mkl_sparse_?_trsv.....	373
mkl_sparse_?_mm.....	375
mkl_sparse_?_trsm.....	379
mkl_sparse_?_add.....	382
mkl_sparse_spmv.....	384
mkl_sparse_?_spmmd.....	385
BLAS-like Extensions.....	387
?axpby.....	388
?gem2vu.....	389
?gem2vc.....	391
?gemmt.....	394
?gemm3m.....	397
?gemm3m_batch.....	400
mkl_?imatcopy.....	404
mkl_?omatcopy.....	407
mkl_?omatcopy2.....	409
mkl_?omatadd.....	412

Chapter 3: LAPACK Routines

Routine Naming Conventions.....	417
Fortran 95 Interface Conventions.....	418
Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib	
Implementation.....	419
Matrix Storage Schemes.....	420
Mathematical Notation.....	420
Error Analysis.....	421

Linear Equation Routines.....	422
Computational Routines.....	422
Routines for Matrix Factorization.....	424
Routines for Solving Systems of Linear Equations.....	464
Routines for Estimating the Condition Number.....	504
Refining the Solution and Estimating Its Error.....	534
Routines for Matrix Inversion.....	599
Routines for Matrix Equilibration.....	627
Driver Routines.....	646
?gesv.....	646
?gesvx.....	650
?gesvxx.....	656
?gbsv.....	664
?gbsvx.....	666
?gbsvxx.....	672
?gtsv.....	680
?gtsvx.....	682
?dtsvb.....	686
?posv.....	688
?posvx.....	691
?posvxx.....	696
?ppsv.....	703
?ppsvx.....	705
?pbsv.....	709
?pbsvx.....	711
?ptsv.....	716
?ptsvx.....	718
?sysv.....	721
?sysv_rook.....	723
?sysvx.....	726
?sysvxx.....	730
?hesv.....	737
?hesv_rook.....	740
?hesvx.....	742
?hesvxx.....	746
?spsv.....	754
?spsvx.....	756
?hpsv.....	759
?hpsvx.....	761
Least Squares and Eigenvalue Problem Routines.....	765
Computational Routines.....	765
Orthogonal Factorizations.....	766
Singular Value Decomposition - LAPACK Computational Routines.....	842
Symmetric Eigenvalue Problems.....	867
Generalized Symmetric-Definite Eigenvalue Problems.....	929
Nonsymmetric Eigenvalue Problems.....	943
Generalized Nonsymmetric Eigenvalue Problems.....	989
Generalized Singular Value Decomposition.....	1023
Cosine-Sine Decomposition.....	1033
Driver Routines.....	1044

Linear Least Squares (LLS) Problems.....	1045
Generalized LLS Problems.....	1057
Symmetric Eigenproblems.....	1063
Nonsymmetric Eigenproblems.....	1140
Singular Value Decomposition - LAPACK Driver Routines.....	1159
Cosine-Sine Decomposition.....	1183
Generalized Symmetric Definite Eigenproblems.....	1193
Generalized Nonsymmetric Eigenproblems.....	1254
Auxiliary Routines.....	1278
?lacgv.....	1291
?lacrm.....	1292
?lactr.....	1293
?laesy.....	1294
?rot.....	1295
?spmv.....	1296
?spr.....	1298
?syconv.....	1299
?symv.....	1301
?syr.....	1302
i?max1.....	1304
?sum1.....	1304
?gbtf2.....	1305
?gebd2.....	1306
?gehd2.....	1308
?gelq2.....	1310
?geql2.....	1311
?geqr2.....	1313
?geqr2p.....	1314
?geqrt2.....	1316
?geqrt3.....	1317
?gerq2.....	1319
?gesc2.....	1320
?getc2.....	1322
?getf2.....	1323
?gtts2.....	1324
?isnan.....	1325
?laisnan.....	1326
?labrd.....	1326
?lacn2.....	1329
?lacon.....	1330
?lacpy.....	1332
?ladiv.....	1333
?lae2.....	1334
?laebz.....	1335
?laed0.....	1339
?laed1.....	1341
?laed2.....	1343
?laed3.....	1345
?laed4.....	1347
?laed5.....	1348

?laed6.....	1349
?laed7.....	1350
?laed8.....	1354
?laed9.....	1357
?laeda.....	1358
?laein.....	1360
?laev2.....	1362
?laexc.....	1364
?lag2.....	1365
?lags2.....	1367
?lagtf.....	1370
?lagtm.....	1372
?lagts.....	1373
?lagv2.....	1375
?lahqr.....	1377
?lahrd.....	1379
?lahr2.....	1381
?laic1.....	1384
?lakf2.....	1386
?lain2.....	1387
?lals0.....	1390
?lalsa.....	1393
?lalsd.....	1396
?lamrg.....	1398
?laneg.....	1399
?langb.....	1400
?lange.....	1402
?langt.....	1403
?lanhs.....	1404
?lansb.....	1405
?lanhb.....	1407
?lansp.....	1408
?lanhp.....	1410
?lanst/?lanht.....	1411
?lansy.....	1412
?lanhe.....	1413
?lantb.....	1415
?lantp.....	1416
?lantr.....	1418
?lanv2.....	1420
?lapll.....	1421
?lapmr.....	1422
?lapmt.....	1423
?lapy2.....	1424
?lapy3.....	1425
?laqgb.....	1425
?laqge.....	1427
?laqhb.....	1429
?laqp2.....	1430
?laqps.....	1432

?laqr0.....	1434
?laqr1.....	1437
?laqr2.....	1438
?laqr3.....	1442
?laqr4.....	1445
?laqr5.....	1448
?laqsb.....	1451
?laqsp.....	1453
?laqsy.....	1454
?laqtr.....	1456
?lar1v.....	1458
?lar2v.....	1461
?laran.....	1462
?larf.....	1463
?larfb.....	1464
?larfg.....	1468
?larfgp.....	1469
?larft.....	1471
?larfx.....	1473
?large.....	1475
?largv.....	1476
?larnd.....	1477
?larnv.....	1478
?laror.....	1479
?larot.....	1482
?larra.....	1485
?larrb.....	1487
?larrc.....	1489
?larrd.....	1490
?larre.....	1493
?larrf.....	1497
?larrj.....	1499
?larrk.....	1501
?larrr.....	1502
?larrv.....	1503
?lartg.....	1507
?lartgp.....	1508
?lartgs.....	1510
?lartv.....	1511
?laruv.....	1512
?larz.....	1513
?larzb.....	1515
?larzt.....	1517
?las2.....	1520
?lascl.....	1520
?lasd0.....	1522
?lasd1.....	1523
?lasd2.....	1526
?lasd3.....	1529
?lasd4.....	1532

?lasd5.....	1533
?lasd6.....	1534
?lasd7.....	1538
?lasd8.....	1542
?lasd9.....	1544
?lasda.....	1546
?lasdq.....	1549
?lasdt.....	1551
?laset.....	1552
?lasq1.....	1553
?lasq2.....	1555
?lasq3.....	1556
?lasq4.....	1557
?lasq5.....	1559
?lasq6.....	1560
?lasr.....	1561
?lasrt.....	1564
?lassq.....	1565
?lasv2.....	1567
?laswp.....	1568
?lasy2.....	1569
?lasyf.....	1571
?lasyf_rook.....	1573
?lahef.....	1575
?lahef_rook.....	1577
?latbs.....	1579
?latm1.....	1581
?latm2.....	1583
?latm3.....	1586
?latm5.....	1589
?latm6.....	1593
?latme.....	1596
?latmr.....	1600
?latdf.....	1608
?latps.....	1609
?latrd.....	1612
?latrs.....	1614
?latrz.....	1618
?lauu2.....	1620
?lauum.....	1621
?orbdb1/?unbdb1.....	1622
?orbdb2/?unbdb2.....	1625
?orbdb3/?unbdb3.....	1628
?orbdb4/?unbdb4.....	1631
?orbdb5/?unbdb5.....	1635
?orbdb6/?unbdb6.....	1637
?org2l/?ung2l.....	1640
?org2r/?ung2r.....	1641
?orgl2/?ungl2.....	1642
?orgr2/?ungr2.....	1644

?orm2l/?unm2l.....	1645
?orm2r/?unm2r.....	1647
?orml2/?unml2.....	1649
?ormr2/?unmr2.....	1651
?ormr3/?unmr3.....	1653
?pbt2.....	1655
?pot2.....	1657
?pts2.....	1658
?rscl.....	1659
?syswapr.....	1660
?heswapr.....	1662
?sygs2/?hegs2.....	1665
?sytd2/?hetd2.....	1666
?sytf2.....	1668
?sytf2_rook.....	1670
?hetf2.....	1671
?hetf2_rook.....	1673
?tgex2.....	1674
?tgsy2.....	1677
?trti2.....	1680
clag2z.....	1681
dlag2s.....	1682
slag2d.....	1683
zlag2c.....	1684
?larfp.....	1685
ila?lc.....	1686
ila?lr.....	1687
?gsvj0.....	1688
?gsvj1.....	1690
?sfrk.....	1693
?hfrk.....	1695
?tfsm.....	1696
?lansf.....	1698
?lanhf.....	1700
?tfttp.....	1701
?tfttr.....	1702
?tpqrt2.....	1704
?tprfb.....	1706
?tpttf.....	1709
?tpttr.....	1711
?trttf.....	1712
?trttp.....	1713
?pstf2.....	1714
dlat2s.....	1716
zlat2c.....	1717
?lacp2.....	1718
?la_gbamv.....	1719
?la_gbrcond.....	1721
?la_gbrcond_c.....	1723
?la_gbrcond_x.....	1725

?la_gbrfsx_extended.....	1726
?la_gbrpvgrw.....	1733
?la_geamv.....	1734
?la_gercond.....	1736
?la_gercond_c.....	1737
?la_gercond_x.....	1739
?la_gerfsx_extended.....	1740
?la_heamv.....	1746
?la_hercond_c.....	1748
?la_hercond_x.....	1749
?la_herfsx_extended.....	1751
?la_herpvgrw.....	1757
?la_lin_berr.....	1758
?la_porcond.....	1759
?la_porcond_c.....	1760
?la_porcond_x.....	1762
?la_porfsx_extended.....	1763
?la_porpvgrw.....	1770
?laqhe.....	1771
?laqhp.....	1772
?larcn.....	1774
?la_gerpvgrw.....	1775
?larscl2.....	1776
?lascl2.....	1777
?la_syamv.....	1778
?la_syrcond.....	1780
?la_syrcond_c.....	1781
?la_syrcond_x.....	1783
?la_syrfsx_extended.....	1784
?la_syrpvgrw.....	1791
?la_wwaddw.....	1792
mkl_?tppack.....	1793
mkl_?tpunpack.....	1795
Utility Functions and Routines.....	1797
ilaver.....	1798
ilaenv.....	1798
iparmq.....	1800
ieeeck.....	1802
?labad.....	1803
?lamch.....	1804
?lamc1.....	1805
?lamc2.....	1805
?lamc3.....	1806
?lamc4.....	1807
?lamc5.....	1808
chla_transtype.....	1808
iladiag.....	1809
ilaprec.....	1810
ilatrans.....	1810
ilauplo.....	1811

xerbla_array.....	1812
Test Functions and Routines.....	1812
?latms.....	1813

Chapter 4: ScaLAPACK Routines

Overview.....	1819
Routine Naming Conventions.....	1821
Computational Routines.....	1822
Linear Equations.....	1822
Routines for Matrix Factorization.....	1823
p?getrf.....	1823
p?gbtrf.....	1825
p?dbtrf.....	1827
p?dttrf.....	1830
p?potrf.....	1832
p?pbtrf.....	1833
p?pttrf.....	1836
Routines for Solving Systems of Linear Equations.....	1838
p?getrs.....	1838
p?gbtrs.....	1840
p?dbtrs.....	1842
p?dttrs.....	1845
p?potrs.....	1847
p?pbtrs.....	1848
p?pttrs.....	1851
p?trtrs.....	1853
Routines for Estimating the Condition Number.....	1855
p?gecon.....	1855
p?pocon.....	1857
p?trcon.....	1860
Refining the Solution and Estimating Its Error.....	1863
p?gerfs.....	1863
p?porfs.....	1866
p?trrfs.....	1870
Routines for Matrix Inversion.....	1873
p?getri.....	1873
p?potri.....	1875
p?trtri.....	1877
Routines for Matrix Equilibration.....	1878
p?geequ.....	1878
p?poequ.....	1880
Orthogonal Factorizations.....	1882
p?geqrf.....	1883
p?geqpf.....	1885
p?orgqr.....	1888
p?ungqr.....	1890
p?ormqr.....	1892
p?unmqr.....	1895
p?gelqf.....	1897
p?orglq.....	1900

p?unglq.....	1902
p?ormlq.....	1903
p?unmlq.....	1906
p?geqlf.....	1909
p?orgql.....	1911
p?ungql.....	1913
p?ormql.....	1915
p?unmql.....	1918
p?gerqf.....	1921
p?orgrq.....	1923
p?ungrq.....	1925
p?ormrq.....	1927
p?unmrq.....	1930
p?tzzzf.....	1933
p?ormrz.....	1936
p?unmrz.....	1939
p?ggqrf.....	1942
p?ggrqf.....	1946
Symmetric Eigenproblems.....	1950
p?sytrd.....	1951
p?ormtr.....	1954
p?hetrd.....	1957
p?unmtr.....	1961
p?stebz.....	1964
p?stein.....	1968
Nonsymmetric Eigenvalue Problems.....	1971
p?gehrd.....	1972
p?ormhr.....	1975
p?unmhr.....	1978
p?lahqr.....	1981
p?hseqr.....	1983
Singular Value Decomposition.....	1986
p?gebrd.....	1986
p?ormbr.....	1990
p?unmbr.....	1994
Generalized Symmetric-Definite Eigen Problems.....	1998
p?sygst.....	1999
p?hegst.....	2001
Driver Routines.....	2002
p?gesv.....	2003
p?gesvx.....	2005
p?gbsv.....	2010
p?dbsv.....	2012
p?dtsv.....	2015
p?posv.....	2017
p?posvx.....	2019
p?pbsv.....	2025
p?ptsv.....	2027
p?gels.....	2030
p?syev.....	2034

p?syevd.....	2036
p?syevr.....	2039
p?syevx.....	2043
p?heev.....	2050
p?heevd.....	2053
p?heevr.....	2055
p?heevx.....	2061
p?gesvd.....	2068
p?sygvx.....	2072
p?hegvx.....	2079

Chapter 5: ScaLAPACK Auxiliary, Utility, and Redistribution/Copy Routines

Auxiliary Routines.....	2089
b?laapp.....	2095
b?laexc.....	2096
b?trexc.....	2098
p?lacgv.....	2100
p?max1.....	2101
pmpcol.....	2102
pmpim2.....	2103
?combamax1.....	2104
p?sum1.....	2105
p?dbtrsv.....	2106
p?dttrsv.....	2108
p?gebal.....	2112
p?gebd2.....	2114
p?gehd2.....	2117
p?gelq2.....	2119
p?geql2.....	2122
p?geqr2.....	2124
p?gerq2.....	2126
p?getf2.....	2128
p?labrd.....	2130
p?lacon.....	2134
p?laconsb.....	2135
p?lcp2.....	2136
p?lcp3.....	2138
p?lacpy.....	2139
p?laevswp.....	2141
p?lahrd.....	2142
p?laiect.....	2145
p?lamve.....	2146
p?lange.....	2148
p?lanhs.....	2149
p?lansy, p?lanhe.....	2151
p?lantr.....	2153
p?lapiv.....	2155
p?laqge.....	2157
p?laqr0.....	2160

p?laqr1.....	2162
p?laqr2.....	2165
p?laqr3.....	2168
p?laqr4.....	2171
p?laqr5.....	2174
p?laqsy.....	2176
p?lared1d.....	2178
p?lared2d.....	2179
p?larf.....	2180
p?larfb.....	2183
p?larfc.....	2187
p?larfg.....	2190
p?larft.....	2191
p?larz.....	2194
p?larzb.....	2197
p?larzc.....	2201
p?larzt.....	2203
p?lascl.....	2206
p?laset.....	2208
p?lasmsub.....	2209
p?lassq.....	2211
p?laswp.....	2212
p?latra.....	2214
p?latrd.....	2215
p?latrs.....	2219
p?latrz.....	2221
p?lauu2.....	2223
p?lauum.....	2225
p?lawil.....	2226
p?org2l/p?ung2l.....	2227
p?org2r/p?ung2r.....	2229
p?orgl2/p?ungl2.....	2231
p?orgr2/p?ungr2.....	2234
p?orm2l/p?unm2l.....	2236
p?orm2r/p?unm2r.....	2239
p?orml2/p?unml2.....	2243
p?ormr2/p?unmr2.....	2246
p?pbtrsv.....	2250
p?pttrsv.....	2254
p?potf2.....	2257
p?rot.....	2259
p?rscl.....	2261
p?sygs2/p?hegs2.....	2262
p?sytd2/p?hetd2.....	2264
p?trord.....	2268
p?trsen.....	2271
p?trti2.....	2276
?lamsh.....	2278
?laqr6.....	2279
?lar1va.....	2282

?laref.....	2285
?larrb2.....	2287
?larrd2.....	2290
?larre2.....	2293
?larre2a.....	2298
?larrf2.....	2302
?larrv2.....	2304
?lasorte.....	2310
?lasrt2.....	2311
?stegr2.....	2312
?stegr2a.....	2315
?stegr2b.....	2319
?stein2.....	2324
?dbtf2.....	2326
?dbtrf.....	2327
?dttrf.....	2328
?dttrsv.....	2329
?pttrsv.....	2331
?steqr2.....	2332
Utility Functions and Routines.....	2334
p?labad.....	2334
p?lachkieee.....	2335
p?lamch.....	2336
p?lasnbt.....	2337
Matrix Redistribution/Copy Routines.....	2338
p?gemr2d.....	2338
p?trmr2d.....	2340

Chapter 6: Sparse Solver Routines

Intel MKL PARDISO - Parallel Direct Sparse Solver Interface.....	2345
pardiso.....	2350
pardisoinit.....	2357
pardiso_64.....	2358
pardiso_getenv, pardiso_setenv.....	2359
mkl_pardiso_pivot.....	2361
pardiso_getdiag.....	2362
pardiso_handle_store.....	2363
pardiso_handle_restore.....	2364
pardiso_handle_delete.....	2365
pardiso_handle_store_64.....	2366
pardiso_handle_restore_64.....	2366
pardiso_handle_delete_64.....	2367
Intel MKL PARDISO Parameters in Tabular Form.....	2368
pardiso iparm Parameter.....	2372
PARDISO_DATA_TYPE.....	2383
Parallel Direct Sparse Solver for Clusters Interface.....	2384
cluster_sparse_solver.....	2385
cluster_sparse_solver iparm Parameter.....	2391
Direct Sparse Solver (DSS) Interface Routines.....	2395
DSS Interface Description.....	2397

DSS Implementation Details.....	2398
DSS Routines.....	2399
dss_create.....	2399
dss_define_structure.....	2400
dss_reorder.....	2402
dss_factor_real, dss_factor_complex.....	2403
dss_solve_real, dss_solve_complex.....	2405
dss_delete.....	2407
dss_statistics.....	2408
mkl_cvt_to_null_terminated_str.....	2411
Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS).....	2411
CG Interface Description.....	2413
FGMRES Interface Description.....	2418
RCI ISS Routines.....	2426
dcg_init.....	2426
dcg_check.....	2427
dcg.....	2428
dcg_get.....	2429
dcgmrhs_init.....	2430
dcgmrhs_check.....	2431
dcgmrhs.....	2432
dcgmrhs_get.....	2434
dfgmres_init.....	2435
dfgmres_check.....	2436
dfgmres.....	2437
dfgmres_get.....	2439
Implementation Details.....	2440
Preconditioners based on Incomplete LU Factorization Technique.....	2440
ILU0 and ILUT Preconditioners Interface Description.....	2442
dcsrilu0.....	2443
dcsrilut.....	2445
Sparse Matrix Checker Routines.....	2449
sparse_matrix_checker.....	2449
sparse_matrix_init.....	2451

Chapter 7: Extended Eigensolver Routines

The FEAST Algorithm.....	2453
Extended Eigensolver Functionality.....	2454
Parallelism in Extended Eigensolver Routines.....	2455
Achieving Performance With Extended Eigensolver Routines.....	2455
Extended Eigensolver Interfaces.....	2456
Extended Eigensolver Naming Conventions.....	2456
feastinit.....	2457
Extended Eigensolver Input Parameters.....	2458
Extended Eigensolver Output Details.....	2459
Extended Eigensolver RCI Routines.....	2460
Extended Eigensolver RCI Interface Description.....	2460
?feast_src/?feast_hrci.....	2462
Extended Eigensolver Predefined Interfaces.....	2465

Matrix Storage.....	2466
?feast_syev/?feast_heev.....	2466
?feast_sygv/?feast_hegv.....	2468
?feast_sbev/?feast_hbev.....	2471
?feast_sbgv/?feast_hbgv.....	2473
?feast_scsrev/?feast_hcsrev.....	2476
?feast_scsrgv/?feast_hcsrgv.....	2478

Chapter 8: Vector Mathematical Functions

Data Types, Accuracy Modes, and Performance Tips.....	2483
Function Naming Conventions.....	2484
Function Interfaces.....	2485
VM Mathematical Functions.....	2485
Pack Functions.....	2485
Unpack Functions.....	2485
Service Functions.....	2485
Input Parameters.....	2486
Output Parameters.....	2486
Vector Indexing Methods.....	2486
Error Diagnostics.....	2487
VM Mathematical Functions.....	2488
Special Value Notations.....	2489
Arithmetic Functions.....	2490
v?Add.....	2490
v?Sub.....	2492
v?Sqr.....	2493
v?Mul.....	2494
v?MulByConj.....	2496
v?Conj.....	2497
v?Abs.....	2498
v?Arg.....	2500
v?LinearFrac.....	2501
Power and Root Functions.....	2503
v?Inv.....	2503
v?Div.....	2504
v?Sqrt.....	2506
v?InvSqrt.....	2508
v?Cbrt.....	2509
v?InvCbrt.....	2510
v?Pow2o3.....	2511
v?Pow3o2.....	2512
v?Pow.....	2513
v?Powx.....	2516
v?Hypot.....	2518
Exponential and Logarithmic Functions.....	2519
v?Exp.....	2520
v?Expn1.....	2522
v?Ln.....	2523
v?Log10.....	2525
v?Log1p.....	2527

Trigonometric Functions.....	2528
v?Cos.....	2528
v?Sin.....	2530
v?SinCos.....	2532
v?CIS.....	2534
v?Tan.....	2535
v?Acos.....	2537
v?Asin.....	2539
v?Atan.....	2541
v?Atan2.....	2542
Hyperbolic Functions.....	2544
v?Cosh.....	2544
v?Sinh.....	2546
v?Tanh.....	2549
v?Acosh.....	2551
v?Asinh.....	2553
v?Atanh.....	2555
Special Functions.....	2557
v?Erf.....	2557
v?Erfc.....	2559
v?CdfNorm.....	2561
v?ErfInv.....	2563
v?ErfcInv.....	2565
v?CdfNormInv.....	2567
v?LGamma.....	2568
v?TGamma.....	2570
Rounding Functions.....	2571
v?Floor.....	2571
v?Ceil.....	2572
v?Trunc.....	2573
v?Round.....	2574
v?NearbyInt.....	2576
v?Rint.....	2577
v?Modf.....	2578
v?Frac.....	2579
VM Pack/Unpack Functions.....	2580
v?Pack.....	2581
v?Unpack.....	2583
VM Service Functions.....	2584
vmlSetMode.....	2585
vmlgetmode.....	2586
vmlSetErrStatus.....	2587
vmlgeterrstatus.....	2588
vmlclearerrstatus.....	2588
vmlSetErrorCallBack.....	2589
vmlGetErrorCallBack.....	2591
vmlClearErrorCallBack.....	2591

Chapter 9: Statistical Functions

Random Number Generators.....	2593
-------------------------------	------

Conventions.....	2594
Mathematical Notation.....	2595
Naming Conventions.....	2596
Basic Generators.....	2599
BRNG Parameter Definition.....	2601
Random Streams.....	2602
Data Types.....	2602
Error Reporting.....	2603
VS RNG Usage Model.....	2604
Service Routines.....	2606
vslNewStream.....	2607
vslNewStreamEx.....	2608
vslNewAbstractStream.....	2609
vslNewAbstractStream.....	2611
vslNewAbstractStream.....	2613
vslDeleteStream.....	2614
vslCopyStream.....	2615
vslCopyStreamState.....	2616
vslSaveStreamF.....	2616
vslLoadStreamF.....	2617
vslSaveStreamM.....	2619
vslLoadStreamM.....	2620
vslGetStreamSize.....	2621
vslLeapfrogStream.....	2621
vslSkipAheadStream.....	2623
vslGetStreamStateBrng.....	2625
vslGetNumRegBrngs.....	2625
Distribution Generators.....	2626
Continuous Distributions.....	2629
Discrete Distributions.....	2657
Advanced Service Routines.....	2675
Data types.....	2675
vslGetBrngProperties.....	2676
Convolution and Correlation.....	2676
Naming Conventions.....	2677
Data Types.....	2678
Parameters.....	2678
Task Status and Error Reporting.....	2680
Task Constructors.....	2681
vslConvNewTask/vslCorrNewTask.....	2682
vslConvNewTask1D/vslCorrNewTask1D.....	2684
vslConvNewTaskX/vslCorrNewTaskX.....	2685
vslConvNewTaskX1D/vslCorrNewTaskX1D.....	2688
Task Editors.....	2690
vslConvSetMode/vslCorrSetMode.....	2691
vslConvSetInternalPrecision/vslCorrSetInternalPrecision.....	2692
vslConvSetStart/vslCorrSetStart.....	2693
vslConvSetDecimation/vslCorrSetDecimation.....	2694
Task Execution Routines.....	2694
vslConvExec/vslCorrExec.....	2695

vslConvExec1D/vslCorrExec1D.....	2697
vslConvExecX/vslCorrExecX.....	2699
vslConvExecX1D/vslCorrExecX1D.....	2702
Task Destructors.....	2704
vslConvDeleteTask/vslCorrDeleteTask.....	2704
Task Copy.....	2705
vslConvCopyTask/vslCorrCopyTask.....	2705
Usage Examples.....	2706
Mathematical Notation and Definitions.....	2709
Data Allocation.....	2710
Summary Statistics.....	2712
Naming Conventions.....	2713
Data Types.....	2713
Parameters.....	2714
Task Status and Error Reporting.....	2714
Task Constructors.....	2718
vslSSNewTask.....	2718
Task Editors.....	2720
vslSSEditTask.....	2721
vslSSEditMoments.....	2730
vslSSEditSums.....	2731
vslSSEditCovCor.....	2732
vslSSEditCP.....	2734
vslSSEditPartialCovCor.....	2735
vslSSEditQuantiles.....	2736
vslSSEditStreamQuantiles.....	2737
vslSSEditPooledCovariance.....	2738
vslSSEditRobustCovariance.....	2740
vslSSEditOutliersDetection.....	2741
vslSSEditMissingValues.....	2743
vslSSEditCorParameterization.....	2746
Task Computation Routines.....	2747
vslSSCompute.....	2750
Task Destructor.....	2751
vslSSDeleteTask.....	2751
Usage Examples.....	2752
Mathematical Notation and Definitions.....	2753
Chapter 10: Fourier Transform Functions	
FFT Functions.....	2760
FFT Interface.....	2761
Computing an FFT.....	2761
Configuration Settings.....	2761
DFTI_PRECISION.....	2764
DFTI_FORWARD_DOMAIN.....	2764
DFTI_DIMENSION, DFTI_LENGTHS.....	2765
DFTI_PLACEMENT.....	2765
DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE.....	2766
DFTI_NUMBER_OF_USER_THREADS.....	2766
DFTI_THREAD_LIMIT.....	2766

DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES.....	2767
DFTI_NUMBER_OF_TRANSFORMS.....	2768
DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE.....	2769
DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE.....	2770
DFTI_PACKED_FORMAT.....	2772
DFTI_WORKSPACE.....	2780
DFTI_COMMIT_STATUS.....	2780
DFTI_ORDERING.....	2781
Descriptor Manipulation Functions.....	2781
DftiCreateDescriptor.....	2781
DftiCommitDescriptor.....	2783
DftiFreeDescriptor.....	2784
DftiCopyDescriptor.....	2785
Descriptor Configuration Functions.....	2786
DftiSetValue.....	2786
DftiGetValue.....	2788
FFT Computation Functions.....	2790
DftiComputeForward.....	2790
DftiComputeBackward.....	2792
Configuring and Computing an FFT in Fortran.....	2795
Status Checking Functions.....	2797
DftiErrorClass.....	2797
DftiErrorMessage.....	2799
Cluster FFT Functions.....	2800
Computing Cluster FFT.....	2800
Distributing Data among Processes.....	2801
Cluster FFT Interface.....	2803
Descriptor Manipulation Functions.....	2803
DftiCreateDescriptorDM.....	2804
DftiCommitDescriptorDM.....	2805
DftiFreeDescriptorDM.....	2805
FFT Computation Functions.....	2806
DftiComputeForwardDM.....	2806
DftiComputeBackwardDM.....	2808
Descriptor Configuration Functions.....	2810
DftiSetValueDM.....	2810
DftiGetValueDM.....	2812
Error Codes.....	2814

Chapter 11: PBLAS Routines

Overview.....	2815
Routine Naming Conventions.....	2816
PBLAS Level 1 Routines.....	2818
p?amax.....	2818
p?asum.....	2819
p?axpy.....	2820
p?copy.....	2822
p?dot.....	2823
p?dotc.....	2825

p?dotu.....	2826
p?nrm2.....	2827
p?scal.....	2828
p?swap.....	2829
PBLAS Level 2 Routines.....	2830
p?gemv.....	2831
p?agemv.....	2834
p?ger.....	2836
p?gerc.....	2838
p?geru.....	2840
p?hemv.....	2842
p?ahemv.....	2844
p?her.....	2846
p?her2.....	2847
p?symv.....	2849
p?asymv.....	2851
p?syr.....	2853
p?syr2.....	2855
p?trmv.....	2857
p?atrmv.....	2859
p?trsv.....	2861
PBLAS Level 3 Routines.....	2863
p?geadd.....	2864
p?tradd.....	2866
p?gemm.....	2868
p?hemm.....	2870
p?herk.....	2872
p?her2k.....	2874
p?symm.....	2877
p?syrk.....	2879
p?syr2k.....	2881
p?tran.....	2884
p?tranu.....	2885
p?tranc.....	2886
p?trmm.....	2888
p?trsm.....	2890

Chapter 12: Partial Differential Equations Support

Trigonometric Transform Routines.....	2893
Transforms Implemented.....	2894
Sequence of Invoking TT Routines.....	2895
Interface Description.....	2896
TT Routines.....	2897
?_init_trig_transform.....	2897
?_commit_trig_transform.....	2898
?_forward_trig_transform.....	2900
?_backward_trig_transform.....	2902
free_trig_transform.....	2904
Common Parameters.....	2905
Implementation Details.....	2908

Fast Poisson Solver Routines	2909
Poisson Solver Implementation.....	2910
Sequence of Invoking Poisson Solver Routines.....	2915
Interface Description.....	2917
Routines for the Cartesian Solver.....	2918
?_init_Helmholtz_2D/?_init_Helmholtz_3D.....	2918
_commit_Helmholtz_2D/?_commit_Helmholtz_3D.....	2920
?_Helmholtz_2D/?_Helmholtz_3D.....	2923
free_Helmholtz_2D/free_Helmholtz_3D.....	2926
Routines for the Spherical Solver.....	2927
?_init_sph_p/?_init_sph_np.....	2927
?_commit_sph_p/?_commit_sph_np.....	2929
?_sph_p/?_sph_np.....	2931
free_sph_p/free_sph_np.....	2933
Common Parameters.....	2934
ipar.....	2934
dpar and spar.....	2938
Caveat on Parameter Modifications.....	2940
Parameters That Define Boundary Conditions.....	2940
Implementation Details.....	2943
Calling PDE Support Routines from Fortran.....	2949

Chapter 13: Nonlinear Optimization Problem Solvers

Organization and Implementation.....	2951
Routine Naming Conventions.....	2952
Nonlinear Least Squares Problem without Constraints.....	2952
?trnlsp_init.....	2953
?trnlsp_check.....	2955
?trnlsp_solve.....	2956
?trnlsp_get.....	2958
?trnlsp_delete.....	2959
Nonlinear Least Squares Problem with Linear (Bound) Constraints.....	2960
?trnlspbc_init.....	2960
?trnlspbc_check.....	2962
?trnlspbc_solve.....	2965
?trnlspbc_get.....	2966
?trnlspbc_delete.....	2967
Jacobian Matrix Calculation Routines.....	2968
?jacobi_init.....	2968
?jacobi_solve.....	2969
?jacobi_delete.....	2970
?jacobi.....	2971
?jacobix.....	2972

Chapter 14: Support Functions

Using a Fortran Interface Module for Support Functions.....	2978
Version Information.....	2979
mkl_get_version_string.....	2979
Threading Control.....	2979
mkl_set_num_threads.....	2981

mkl_domain_set_num_threads.....	2981
mkl_set_num_threads_local.....	2982
mkl_set_dynamic.....	2984
mkl_get_max_threads.....	2985
mkl_domain_get_max_threads.....	2986
mkl_get_dynamic.....	2987
Error Handling.....	2988
Error Handling for Linear Algebra Routines.....	2988
xerbla.....	2988
pxerbla.....	2989
Handling Fatal Errors.....	2990
mkl_set_exit_handler.....	2990
Character Equality Testing.....	2991
lsame.....	2991
lsamen.....	2991
Timing.....	2992
second/dsecnd.....	2992
mkl_get_cpu_clocks.....	2993
mkl_get_cpu_frequency.....	2993
mkl_get_max_cpu_frequency.....	2994
mkl_get_clocks_frequency.....	2994
Memory Management.....	2995
mkl_free_buffers.....	2995
mkl_thread_free_buffers.....	2996
mkl_disable_fast_mm.....	2996
mkl_mem_stat.....	2997
mkl_peak_mem_usage.....	2998
mkl_malloc.....	2999
mkl_calloc.....	2999
mkl_realloc.....	3000
mkl_free.....	3001
mkl_set_memory_limit.....	3002
Usage Examples for the Memory Functions.....	3003
Single Dynamic Library Control.....	3005
mkl_set_interface_layer.....	3005
mkl_set_threading_layer.....	3006
mkl_set_xerbla.....	3007
mkl_set_progress.....	3008
mkl_set_pardiso_pivot.....	3009
Intel Many Integrated Core Architecture Support.....	3009
mkl_mic_enable.....	3010
mkl_mic_disable.....	3010
mkl_mic_get_device_count.....	3011
mkl_mic_set_workdivision.....	3012
mkl_mic_get_workdivision.....	3013
mkl_mic_set_max_memory.....	3015
mkl_mic_free_memory.....	3016
mkl_mic_register_memory.....	3017
mkl_mic_set_device_num_threads.....	3018
mkl_mic_set_resource_limit.....	3019

mkl_mic_get_resource_limit.....	3022
mkl_mic_set_offload_report.....	3023
mkl_mic_set_flags.....	3024
mkl_mic_get_flags.....	3024
mkl_mic_get_status.....	3025
mkl_mic_clear_status.....	3027
mkl_mic_get_meminfo.....	3028
mkl_mic_get_cpuinfo.....	3029
Conditional Numerical Reproducibility Control.....	3031
mkl_cbwr_set.....	3031
mkl_cbwr_get.....	3032
mkl_cbwr_get_auto_branch.....	3033
Named Constants for CNR Control.....	3034
Reproducibility Conditions.....	3035
Usage Examples for CNR Support Functions.....	3036
Miscellaneous.....	3036
mkl_progress.....	3036
mkl_enable_instructions.....	3037
mkl_set_env_mode.....	3039
mkl_verbose.....	3039
mkl_set_mpi	3040
mkl_finalize.....	3042

Chapter 15: BLACS Routines

Matrix Shapes.....	3043
Repeatability and Coherence.....	3044
BLACS Combine Operations.....	3047
?gamx2d.....	3048
?gamn2d.....	3049
?gsum2d.....	3051
BLACS Point To Point Communication.....	3052
?gesd2d.....	3054
?trsd2d.....	3054
?gerv2d.....	3055
?trrv2d.....	3056
BLACS Broadcast Routines.....	3056
?gebs2d.....	3058
?trbs2d.....	3058
?gebr2d.....	3059
?trbr2d.....	3060
BLACS Support Routines.....	3061
Initialization Routines.....	3061
blacs_pinfo.....	3061
blacs_setup.....	3062
blacs_get.....	3062
blacs_set.....	3063
blacs_gridinit.....	3065
blacs_gridmap.....	3066
Destruction Routines.....	3067
blacs_freebuff.....	3068

blacs_gridexit.....	3068
blacs_abort.....	3068
blacs_exit.....	3069
Informational Routines.....	3069
blacs_gridinfo.....	3070
blacs_pnum.....	3070
blacs_pcoord.....	3070
Miscellaneous Routines.....	3071
blacs_barrier.....	3071
Examples of BLACS Routines Usage.....	3072

Chapter 16: Data Fitting Functions

Naming Conventions.....	3083
Data Types.....	3084
Mathematical Conventions for Data Fitting Functions.....	3084
Data Fitting Usage Model.....	3087
Data Fitting Usage Examples.....	3087
Task Status and Error Reporting.....	3087
Task Creation and Initialization Routines.....	3090
df?newtask1d.....	3090
Task Configuration Routines.....	3092
df?editppspline1d.....	3092
df?editptr.....	3099
dfeditval.....	3100
df?editidxptr.....	3102
df?queryptr.....	3103
dfqueryval.....	3104
df?queryidxptr.....	3105
Computational Routines.....	3106
df?construct1d.....	3107
df?interpolate1d/df?interpolateex1d.....	3108
df?integrate1d/df?integrateex1d.....	3113
df?searchcells1d/df?searchcellsex1d.....	3117
df?interpcallback.....	3119
df?integrcallback.....	3120
df?searchcellscallback.....	3121
Task Destructors.....	3123
dfdeletetask.....	3123

Appendix A: Linear Solvers Basics

Sparse Linear Systems.....	3125
Matrix Fundamentals.....	3125
Direct Method.....	3126
Sparse Matrix Storage Formats.....	3130
DSS Symmetric Matrix Storage.....	3130
DSS Nonsymmetric Matrix Storage.....	3131
DSS Structurally Symmetric Matrix Storage.....	3132
DSS Distributed Symmetric Matrix Storage.....	3133
Sparse BLAS CSR Matrix Storage Format.....	3134
Sparse BLAS CSC Matrix Storage Format.....	3136

Sparse BLAS Coordinate Matrix Storage Format.....	3136
Sparse BLAS Diagonal Matrix Storage Format.....	3137
Sparse BLAS Skyline Matrix Storage Format.....	3138
Sparse BLAS BSR Matrix Storage Format.....	3139

Appendix B: Routine and Function Arguments

Vector Arguments in BLAS.....	3143
Vector Arguments in VM.....	3144
Matrix Arguments.....	3145

Appendix C: FFTW Interface to Intel® Math Kernel Library

Notational Conventions	3151
FFTW2 Interface to Intel® Math Kernel Library	3151
Wrappers Reference.....	3151
One-dimensional Complex-to-complex FFTs	3151
Multi-dimensional Complex-to-complex FFTs.....	3152
One-dimensional Real-to-half-complex/Half-complex-to-real FFTs.....	3152
Multi-dimensional Real-to-complex/Complex-to-real FFTs.....	3152
Multi-threaded FFTW.....	3153
FFTW Support Functions.....	3153
Calling Wrappers from Fortran.....	3154
Limitations of the FFTW2 Interface to Intel MKL.....	3155
Installation.....	3155
Creating the Wrapper Library.....	3155
Application Assembling	3156
Running Examples	3156
FFTW3 Interface to Intel® Math Kernel Library.....	3157
Using FFTW3 Wrappers.....	3157
Calling Wrappers from Fortran.....	3159
Building Your Own Wrapper Library.....	3159
Building an Application.....	3160
Running Examples	3160
MPI FFTW Wrappers.....	3160
Building Your Own Wrapper Library.....	3161
Building an Application.....	3161
Running Examples.....	3161

Appendix D: Code Examples

BLAS Code Examples.....	3163
Fourier Transform Functions Code Examples.....	3166
FFT Code Examples.....	3166
Examples of Using OpenMP* Threading for FFT Computation.....	3170
Examples for Cluster FFT Functions.....	3173
Auxiliary Data Transformations.....	3174

Bibliography

Glossary

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Cilk, Intel, the Intel logo, Intel Atom, Intel Core, Intel Inside, Intel NetBurst, Intel SpeedStep, Intel vPro, Intel Xeon Phi, Intel XScale, Itanium, MMX, Pentium, Thunderbolt, Ultrabook, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party Content

Intel® Math Kernel Library (Intel® MKL) includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.
- Sections on the Linear Algebra PACKage (LAPACK) routines include derivative work portions that have been copyrighted:

© 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.

- Intel MKL supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines under the following license:

Copyright © 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2000-2011 The University of California Berkeley. All rights reserved.

Copyright © 2006-2012 The University of Colorado Denver. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/lapack/index.html>. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

- The original versions of the Basic Linear Algebra Subprograms (BLAS) from which the respective part of Intel® MKL was derived can be obtained from <http://www.netlib.org/blas/index.html>.
- XBLAS is distributed under the following copyright:

Copyright © 2008-2009 The University of California Berkeley. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- The original versions of the Basic Linear Algebra Communication Subprograms (BLACS) from which the respective part of Intel MKL was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.
- The original versions of Scalable LAPACK (ScaLAPACK) from which the respective part of Intel® MKL was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.
- The original versions of the Parallel Basic Linear Algebra Subprograms (PBLAS) routines from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/scalapack/html/pblas_qref.html.
- PARDISO (PARallel DIrect SOLver)* in Intel® MKL was originally developed by the Department of Computer Science at the University of Basel (<http://www.unibas.ch>). It can be obtained at <http://www.pardiso-project.org>.
- The Extended Eigensolver functionality is based on the Feast solver package and is distributed under the following license:

Copyright © 2009, The Regents of the University of Massachusetts, Amherst.
Developed by E. Polizzi
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some Fast Fourier Transform (FFT) functions in this release of Intel® MKL have been generated by the SPIRAL software generation system (<http://www.spiral.net/>) under license from Carnegie Mellon University. The authors of SPIRAL are Markus Puschel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo.
- Open MPI is distributed under the New BSD license, listed below.

Most files in this release are marked with the copyrights of the organizations who have edited them. The copyrights below are in no particular order and generally reflect members of the Open MPI core team who have contributed code to this release. The copyrights for code used under license from other parties are included in the corresponding files.

Copyright © 2004-2010 The Trustees of Indiana University and Indiana University Research and Technology Corporation. All rights reserved.

Copyright © 2004-2010 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2004-2010 High Performance Computing Center Stuttgart, University of Stuttgart. All rights reserved.

Copyright © 2004-2008 The Regents of the University of California. All rights reserved.

Copyright © 2006-2010 Los Alamos National Security, LLC. All rights reserved.

Copyright © 2006-2010 Cisco Systems, Inc. All rights reserved.

Copyright © 2006-2010 Voltaire, Inc. All rights reserved.

Copyright © 2006-2011 Sandia National Laboratories. All rights reserved.

Copyright © 2006-2010 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Copyright © 2006-2010 The University of Houston. All rights reserved.

Copyright © 2006-2009 Myricom, Inc. All rights reserved.

Copyright © 2007-2008 UT-Battelle, LLC. All rights reserved.

Copyright © 2007-2010 IBM Corporation. All rights reserved.

Copyright © 1998-2005 Forschungszentrum Juelich, Juelich Supercomputing Centre, Federal Republic of Germany

Copyright © 2005-2008 ZIH, TU Dresden, Federal Republic of Germany

Copyright © 2007 Evergrid, Inc. All rights reserved.

Copyright © 2008 Chelsio, Inc. All rights reserved.

Copyright © 2008-2009 Institut National de Recherche en Informatique. All rights reserved.

Copyright © 2007 Lawrence Livermore National Security, LLC. All rights reserved.

Copyright © 2007-2009 Mellanox Technologies. All rights reserved.

Copyright © 2006-2010 QLogic Corporation. All rights reserved.

Copyright © 2008-2010 Oak Ridge National Labs. All rights reserved.

Copyright © 2006-2010 Oracle and/or its affiliates. All rights reserved.

Copyright © 2009 Bull SAS. All rights reserved.

Copyright © 2010 ARM Ltd. All rights reserved.

Copyright © 2010-2011 Alex Brick . All rights reserved.

Copyright © 2012 The University of Wisconsin-La Crosse. All rights reserved.

Copyright © 2013-2014 Intel, Inc. All rights reserved.

Copyright © 2011-2014 NVIDIA Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- The Safe C Library is distributed under the following copyright:

Copyright (c)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- HPL Copyright Notice and Licensing Terms

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1.** Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2.** Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
- 3.** All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories.
- 4.** The name of the University, the name of the Laboratory, or the names of its contributors may not be used to endorse or promote products derived from this software without specific written permission.

Copyright© 2016, Intel Corporation. All rights reserved.

Introducing the Intel® Math Kernel Library

The Intel® Math Kernel Library (Intel® MKL) improves performance with math routines for software applications that solve large computational problems. Intel MKL provides BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

NOTE

It is your responsibility when using Intel MKL to ensure that input data has the required format and does not contain invalid characters. These can cause unexpected behavior of the library.

Intel MKL is optimized for the latest Intel processors, including processors with multiple cores (see the *Intel MKL Release Notes* for the full list of supported processors). Intel MKL also performs well on non-Intel processors.

For more details about functionality provided by Intel MKL, see the [Function Domains](#) section.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Getting Help and Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://www.intel.com/software/products/support>.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit <http://www.intel.com/software/products/>).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit <http://www.intel.com/software/products/support>.

What's New

This Reference Manual documents Intel® Math Kernel Library (Intel® MKL) 11.3 Update 2 release for the Fortran interface.

The following function domains were updated with new functions, enhancements to the existing functionality, or improvements to the existing documentation:

- The `mkl_finalize` function has been added to Support Functions to help avoid resource leaks when a third-party shared library is loaded and unloaded multiple times. See [mkl_finalize](#) for more details.
- The quality of math equations has been improved for some topics, including all [Fourier Transform](#) and [Partial Differential Equations Support](#) functions.
- Functionality has been added to [Summary Statistics](#) to sort the datasets by components of a random vector ξ .

The manual has also been updated to reflect other enhancements to the product, and minor improvements have been made.

Additionally, minor updates have been made to correct errors in the manual.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Notational Conventions

This manual uses the following terms to refer to operating systems:

Windows* OS	This term refers to information that is valid on all supported Windows* operating systems.
Linux* OS	This term refers to information that is valid on all supported Linux* operating systems.
OS X*	This term refers to information that is valid on Intel®-based systems running the OS X* operating system.

This manual uses the following notational conventions:

- Routine name shorthand (for example, `?ungqr` instead of `cungqr/zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, names that contain a question mark "?" represent groups of routines with similar functionality. Each group typically consists of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex. The question mark is used to indicate any or all possible varieties of a function; for example:

<code>?swap</code>	Refers to all four data types of the vector-vector <code>?swap</code> routine: <code>sswap</code> , <code>dswap</code> , <code>cswap</code> , and <code>zswap</code> .
--------------------	--

Font Conventions

The following font conventions are used:

<code>UPPERCASE COURIER</code>	Data type used in the description of input and output parameters for Fortran interface. For example, <code>CHARACTER*1</code> .
<code>lowercase courier</code>	Code examples: <code>a(k+i,j) = matrix(i,j)</code>
<i>lowercase courier italic</i>	Variables in arguments and parameters description. For example, <i>incx</i> .
*	Used as a multiplication symbol in code examples and equations and where required by the programming language syntax.

Function Domains

es

BLAS Routines

The BLAS routines and functions are divided into the following groups according to the operations they perform:

- [BLAS Level 1 Routines](#) perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- [BLAS Level 2 Routines](#) perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [BLAS Level 3 Routines](#) perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® MKL also supports the Fortran 95 interface to the BLAS routines.

Starting from release 10.1, a number of [BLAS-like Extensions](#) are added to enable the user to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations.

Sparse BLAS Routines

The [Sparse BLAS Level 1 Routines and Functions](#) and [Sparse BLAS Level 2 and Level 3 Routines](#) routines and functions operate on sparse vectors and matrices. These routines perform vector operations similar to the BLAS Level 1, 2, and 3 routines. The Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel MKL also supports Fortran 95 interface to Sparse BLAS routines.

LAPACK Routines

The Intel® Math Kernel Library fully supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/lapack/index.html>. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see [LAPACK Routines: Linear Equations](#)).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see [LAPACK Routines: Least Squares and Eigenvalue Problems](#)).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see [LAPACK Auxiliary Routines](#) and [LAPACK Utility Functions and Routines](#)).

Starting from release 8.0, Intel MKL also supports the Fortran 95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

ScaLAPACK Routines

The ScaLAPACK package (provided only for Intel® 64 and Intel® Many Integrated Core architectures, see [Chapter 6](#) and [Chapter 7](#)) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley.

The Intel MKL version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI as well as Intel MPI.

PBLAS Routines

The PBLAS routines perform operations with distributed vectors and matrices.

- [PBLAS Level 1 Routines](#) perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- [PBLAS Level 2 Routines](#) perform distributed matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [PBLAS Level 3 Routines](#) perform distributed matrix-matrix operations, such as matrix-matrix multiplication, rank- k update, and solution of triangular systems.

Intel MKL provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (part of the ScaLAPACK package, see http://www.netlib.org/scalapack/html/pblas_qref.html).

Sparse Solver Routines

Direct sparse solver routines in Intel MKL (see [Chapter 8](#)) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel MKL subroutines can solve both positive-definite and indefinite systems. Intel MKL includes a solver based on the PARDISO* sparse solver, referred to as Intel MKL PARDISO, as well as an alternative set of user callable direct sparse solver routines.

If you use the Intel MKL PARDISO sparse solver, please cite:

O.Schenk and K.Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. *J. of Future Generation Computer Systems*, 20(3):475-487, 2004.

Intel MKL provides also an iterative sparse solver (see [Chapter 8](#)) that uses Sparse BLAS level 2 and 3 routines and works with different sparse data formats.

Extended Eigensolver Routines

The [Extended Eigensolver RCI Routines](#) is a set of high-performance numerical routines for solving standard ($Ax = \lambda x$) and generalized ($Ax = \lambda Bx$) eigenvalue problems, where A and B are symmetric or Hermitian. It yields all the eigenvalues and eigenvectors within a given search interval. It is based on the Feast algorithm, an innovative fast and stable numerical algorithm presented in [[Polizzi09](#)], which deviates fundamentally from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [[Bai00](#)]) or other Davidson-Jacobi techniques [[Sleijpen96](#)]. The Feast algorithm is inspired by the density-matrix representation and contour integration technique in quantum mechanics.

It is free from orthogonalization procedures. Its main computational tasks consist of solving very few inner independent linear systems with multiple right-hand sides and one reduced eigenvalue problem orders of magnitude smaller than the original one. The Feast algorithm combines simplicity and efficiency and offers many important capabilities for achieving high performance, robustness, accuracy, and scalability on parallel architectures. This algorithm is expected to significantly augment numerical performance in large-scale modern applications.

Some of the characteristics of the Feast algorithm [[Polizzi09](#)] are:

- Converges quickly in 2-3 iterations with very high accuracy
- Naturally captures all eigenvalue multiplicities
- No explicit orthogonalization procedure
- Can reuse the basis of pre-computed subspace as suitable initial guess for performing outer-refinement iterations

This capability can also be used for solving a series of eigenvalue problems that are close one another.

- The number of internal iterations is independent of the size of the system and the number of eigenpairs in the search interval
- The inner linear systems can be solved either iteratively (even with modest relative residual error) or directly

VM Functions

The Vector Mathematics functions (see [Chapter 9](#)) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic, etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VM include nonlinear programming software, integrals computation, and many others. VM provides interfaces both for Fortran and C languages.

Statistical Functions

Vector Statistics (VS) contains three sets of functions (see [Chapter 10](#)) providing:

- Pseudorandom, quasi-random, and non-deterministic random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, the VS subroutines use calls to highly optimized Basic Random Number Generators (BRNGs) and a set of vector mathematical functions.
- A wide variety of convolution and correlation operations.
- Initial statistical analysis of raw single and double precision multi-dimensional datasets.

Fourier Transform Functions

The Intel® MKL multidimensional Fast Fourier Transform (FFT) functions with mixed radix support (see [Chapter 11](#)) provide uniformity of discrete Fourier transform computation and combine functionality with ease of use. Both Fortran and C interface specification are given. There is also a cluster version of FFT functions, which runs on distributed-memory architectures and is provided only for Intel® 64 and Intel® Many Integrated Core architectures.

The FFT functions provide fast computation via the FFT algorithms for arbitrary lengths. See *the Intel® MKL User's Guide* for the specific radices supported.

Partial Differential Equations Support

Intel® MKL provides tools for solving Partial Differential Equations (PDE) (see [Chapter 13](#)). These tools are Trigonometric Transform interface routines and Poisson Solver.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the Intel MKL Poisson Solver. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.

The Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on the Intel MKL FFT interface (refer to [Chapter 11](#)), optimized for Intel® processors.

Nonlinear Optimization Problem Solvers

Intel® MKL provides Nonlinear Optimization Problem Solver routines (see [Chapter 14](#)) that can be used to solve nonlinear least squares problems with or without linear (bound) constraints through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.

Support Functions

The Intel® MKL support functions (see [Chapter 15](#)) are used to support the operation of the Intel MKL software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

Starting from release 10.0, the Intel MKL support functions provide additional threading control.

Starting from release 10.1, Intel MKL selectively supports a *Progress Routine* feature to track progress of a lengthy computation and/or interrupt the computation using a callback function mechanism. The user application can define a function called `mkl_progress` that is regularly called from the Intel MKL routine supporting the progress routine feature. See [the Progress Routines](#) section in [Chapter 15](#) for reference. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not.

BLACS Routines

The Intel® Math Kernel Library implements routines from the BLACS (Basic Linear Algebra Communication Subprograms) package (see [Chapter 16](#)) that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The original versions of BLACS from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

Data Fitting Functions

The Data Fitting component includes a set of highly-optimized implementations of algorithms for the following spline-based computations:

- spline construction
- interpolation including computation of derivatives and integration
- search

The algorithms operate on single and double vector-valued functions set in the points of the given partition. You can use Data Fitting algorithms in applications that are based on data approximation. See [Data Fitting Functions](#) for more information.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Performance Enhancements

The Intel® Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs

- Blocking of data to improve data reuse opportunities
- Copying to reduce chances of data eviction from cache
- Data prefetching to help hide memory latency
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines
- Use of hardware features such as the SIMD arithmetic units, where appropriate

These are techniques from which the arithmetic code benefits the most.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Parallelism

In addition to the performance enhancements discussed above, Intel® MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel MKL functions (except for the deprecated `?lacon` LAPACK routine) because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe.
- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for *QR* factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see *Intel® MKL User's Guide*.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BLAS and Sparse BLAS Routines

This chapter describes the Intel® Math Kernel Library implementation of the BLAS and Sparse BLAS routines, and BLAS-like extensions. The routine descriptions are arranged in several sections:

- [BLAS Level 1 Routines](#) (vector-vector operations)
- [BLAS Level 2 Routines](#) (matrix-vector operations)
- [BLAS Level 3 Routines](#) (matrix-matrix operations)
- [Sparse BLAS Level 1 Routines](#) (vector-vector operations).
- [Sparse BLAS Level 2 and Level 3 Routines](#) (matrix-vector and matrix-matrix operations)
- [BLAS-like Extensions](#)

Each section presents the routine and function group descriptions in alphabetical order by routine or function group name; for example, the `?asum` group, the `?axpy` group. The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`.

In BLAS Level 1 groups `i?amax` and `i?amin`, an "i" is placed before the data-type indicator and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

BLAS Routines

Routine Naming Conventions

BLAS routine names have the following structure:

`<character> <name> <mod> ()`

The `<character>` field indicates the data type:

<code>s</code>	real, single precision
<code>c</code>	complex, single precision
<code>d</code>	real, double precision
<code>z</code>	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The `<name>` field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, `<name>` reflects the matrix argument type:

<code>ge</code>	general matrix
<code>gb</code>	general band matrix
<code>sy</code>	symmetric matrix

sp	symmetric matrix (packed storage)
sb	symmetric band matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	Hermitian band matrix
tr	triangular matrix
tp	triangular matrix (packed storage)
tb	triangular band matrix.

The `<mod>` field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the `<mod>` field:

c	conjugated vector
u	unconjugated vector
g	Givens rotation construction
m	modified Givens rotation
mg	modified Givens rotation construction

BLAS level 2 names can have the following characters in the `<mod>` field:

mv	matrix-vector product
sv	solving a system of linear equations with a single unknown vector
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the `<mod>` field:

mm	matrix-matrix product
sm	solving a system of linear equations with multiple unknown vectors
rk	rank- <i>k</i> update of a matrix
r2k	rank-2 <i>k</i> update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

ddot	<code><d> <dot></code> : double-precision real vector-vector dot product
cdotc	<code><c> <dot> <c></code> : complex vector-vector dot product, conjugated
scasum	<code><sc> <asum></code> : sum of magnitudes of vector elements, single precision real output and single precision complex input
cdotu	<code><c> <dot> <u></code> : vector-vector dot product, unconjugated, complex
sgemv	<code><s> <ge> <mv></code> : matrix-vector product, general matrix, single precision

`ztrmm` `<z>` `<tr>` `<mm>`: matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see [Naming Conventions](#).

Fortran 95 Interface Conventions

Fortran 95 interface to BLAS and Sparse BLAS Level 1 routines is implemented through wrappers that call respective FORTRAN 77 routines. This interface uses such features of Fortran 95 as assumed-shape arrays and optional arguments to provide simplified calls to BLAS and Sparse BLAS Level 1 routines with fewer parameters.

NOTE

For BLAS, Intel MKL offers two types of Fortran 95 interfaces:

- using `mkl_blas.fi` only through `include 'mkl.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
 - using `blas.f90` that includes improved interfaces. This file is used to generate the module files `blas95.mod` and `f95_precision.mod`. See also section "Fortran 95 interfaces and wrappers to LAPACK and BLAS" of *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the BLAS interface: `use blas95` and `use f95_precision`.
-

The main conventions used in Fortran 95 interface are as follows:

- The names of parameters used in Fortran 95 interface are typically the same as those used for the respective generic (FORTRAN 77) interface. In rare cases formal argument names may be different.
- Some input parameters such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.
- A parameter can be skipped if its value is completely defined by the presence or absence of another parameter in the calling sequence, and the restored value is the only meaningful value for the skipped parameter.
- Parameters specifying the increment values `incx` and `incy` are skipped. In most cases their values are equal to 1. In Fortran 95 an increment with different value can be directly established in the corresponding parameter.
- Some generic parameters are declared as optional in Fortran 95 interface and may or may not be present in the calling sequence. A parameter can be declared optional if it satisfies one of the following conditions:
 1. It can take only a few possible values. The default value of such parameter typically is the first value in the list; all exceptions to this rule are explicitly stated in the routine description.
 2. It has a natural default value.

Optional parameters are given in square brackets in Fortran 95 call syntax.

The particular rules used for reconstructing the values of omitted optional parameters are specific for each routine and are detailed in the respective "Fortran 95 Notes" subsection at the end of routine specification section. If this subsection is omitted, the Fortran 95 interface for the given routine does not differ from the corresponding FORTRAN 77 interface.

Note that this interface is not implemented in the current version of Sparse BLAS Level 2 and Level 3 routines.

Matrix Storage Schemes

Matrix arguments of BLAS routines can use the following storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array a , with the matrix element A_{ij} stored in the array element $a(i, j)$.

- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see [Matrix Arguments](#) in Appendix B.

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. [Table “BLAS Level 1 Routine Groups and Their Data Types”](#) lists the BLAS Level 1 routine and function groups and the data types associated with them.

BLAS Level 1 Routine and Function Groups and Their Data Types

Routine or Function Group	Data Types	Description
<code>?asum</code>	s, d, sc, dz	Sum of vector magnitudes (functions)
<code>?axpy</code>	s, d, c, z	Scalar-vector product (routines)
<code>?copy</code>	s, d, c, z	Copy vector (routines)
<code>?dot</code>	s, d	Dot product (functions)
<code>?sdot</code>	sd, d	Dot product with double precision (functions)
<code>?dotc</code>	c, z	Dot product conjugated (functions)
<code>?dotu</code>	c, z	Dot product unconjugated (functions)
<code>?norm2</code>	s, d, sc, dz	Vector 2-norm (Euclidean norm) (functions)
<code>?rot</code>	s, d, cs, zd	Plane rotation of points (routines)
<code>?rotg</code>	s, d, c, z	Generate Givens rotation of points (routines)
<code>?rotm</code>	s, d	Modified Givens plane rotation of points (routines)
<code>?rotmg</code>	s, d	Generate modified Givens plane rotation of points (routines)
<code>?scal</code>	s, d, c, z, cs, zd	Vector-scalar product (routines)
<code>?swap</code>	s, d, c, z	Vector-vector swap (routines)
<code>i?amax</code>	s, d, c, z	Index of the maximum absolute value element of a vector (functions)
<code>i?amin</code>	s, d, c, z	Index of the minimum absolute value element of a vector (functions)
<code>?cabs1</code>	s, d	Auxiliary functions, compute the absolute value of a complex number of single or double precision

`?asum`

Computes the sum of magnitudes of the vector elements.

Syntax

```
res = sasum(n, x, incx)
```

```

res = scasum(n, x, incx)
res = dasum(n, x, incx)
res = dzasum(n, x, incx)
res = asum(x)

```

Include Files

- `mkl.fi, blas.f90`

Description

The `?asum` routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

$$res = |Re x_1| + |Im x_1| + |Re x_2| + |Im x_2| + \dots + |Re x_n| + |Im x_n|,$$

where x is a vector with n elements.

Input Parameters

n	INTEGER. Specifies the number of elements in vector x .
x	REAL for <code>sasum</code> DOUBLE PRECISION for <code>dasum</code> COMPLEX for <code>scasum</code> DOUBLE COMPLEX for <code>dzasum</code> Array, size at least $(1 + (n-1) * abs(incx))$.
$incx$	INTEGER. Specifies the increment for indexing vector x .

Output Parameters

res	REAL for <code>sasum</code> DOUBLE PRECISION for <code>dasum</code> REAL for <code>scasum</code> DOUBLE PRECISION for <code>dzasum</code> Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.
-------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `asum` interface are the following:

x	Holds the array of size n .
-----	-------------------------------

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

```
call saxpy(n, a, x, incx, y, incy)
call daxpy(n, a, x, incx, y, incy)
call caxpy(n, a, x, incx, y, incy)
call zaxpy(n, a, x, incx, y, incy)
call axpy(x, y [,a])
```

Include Files

- mkl.fi, blas.f90

Description

The ?axpy routines perform a vector-vector operation defined as

$$y := a*x + y$$

where:

a is a scalar

x and y are vectors each with a number of elements that equals n .

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
a	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Specifies the scalar a .
x	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, size at least $(1 + (n-1)*abs(incx))$.
$incx$	INTEGER. Specifies the increment for the elements of x .
y	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy

Array, size at least $(1 + (n-1) * \text{abs}(incy))$.

incy INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y Contains the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `axpy` interface are the following:

x Holds the array of size *n*.

y Holds the array of size *n*.

a The default value is 1.

?copy

Copies vector to another vector.

Syntax

```
call scopy(n, x, incx, y, incy)
```

```
call dcopy(n, x, incx, y, incy)
```

```
call ccopy(n, x, incx, y, incy)
```

```
call zcopy(n, x, incx, y, incy)
```

```
call copy(x, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?copy` routines perform a vector-vector operation defined as

```
y = x,
```

where *x* and *y* are vectors.

Input Parameters

n INTEGER. Specifies the number of elements in vectors *x* and *y*.

x REAL **for** `scopy`
 DOUBLE PRECISION **for** `dcopy`
 COMPLEX **for** `ccopy`
 DOUBLE COMPLEX **for** `zcopy`

	Array, size at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for <i>scopy</i> DOUBLE PRECISION for <i>dcopy</i> COMPLEX for <i>ccopy</i> DOUBLE COMPLEX for <i>zcopy</i> Array, size at least $(1 + (n-1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>y</i>	Contains a copy of the vector <i>x</i> if <i>n</i> is positive. Otherwise, parameters are unaltered.
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `copy` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .

?dot

Computes a vector-vector dot product.

Syntax

```
res = sdot(n, x, incx, y, incy)
```

```
res = ddot(n, x, incx, y, incy)
```

```
res = dot(x, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?dot` routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^n x_i * y_i,$$

where x_i and y_i are elements of vectors *x* and *y*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>x</i>	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Array, size at least $(1+(n-1)*abs(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Array, size at least $(1+(n-1)*abs(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>res</i>	REAL for <code>sdot</code> DOUBLE PRECISION for <code>ddot</code> Contains the result of the dot product of <i>x</i> and <i>y</i> , if <i>n</i> is positive. Otherwise, <i>res</i> contains 0.
------------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `dot` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .

?sdot

Computes a vector-vector dot product with double precision.

Syntax

```
res = sdsdot(n, sb, sx, incx, sy, incy)
res = dsdot(n, sx, incx, sy, incy)
res = sdot(sx, sy)
res = sdot(sx, sy, sb)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?sdot` routines compute the inner product of two vectors with double precision. Both routines use double precision accumulation of the intermediate results, but the `sdsdot` routine outputs the final result in single precision, whereas the `dsdot` routine outputs the double precision result. The function `sdsdot` also adds scalar value `sb` to the inner product.

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in the input vectors <code>sx</code> and <code>sy</code> .
<code>sb</code>	REAL. Single precision scalar to be added to inner product (for the function <code>sdsdot</code> only).
<code>sx, sy</code>	REAL. Arrays, size at least $(1+(n-1)*abs(incx))$ and $(1+(n-1)*abs(incy))$, respectively. Contain the input single precision vectors.
<code>incx</code>	INTEGER. Specifies the increment for the elements of <code>sx</code> .
<code>incy</code>	INTEGER. Specifies the increment for the elements of <code>sy</code> .

Output Parameters

<code>res</code>	REAL for <code>sdsdot</code> DOUBLE PRECISION for <code>dsdot</code> Contains the result of the dot product of <code>sx</code> and <code>sy</code> (with <code>sb</code> added for <code>sdsdot</code>), if <code>n</code> is positive. Otherwise, <code>res</code> contains <code>sb</code> for <code>sdsdot</code> and 0 for <code>dsdot</code> .
------------------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `sdot` interface are the following:

<code>sx</code>	Holds the vector with the number of elements <code>n</code> .
<code>sy</code>	Holds the vector with the number of elements <code>n</code> .

NOTE

Note that scalar parameter `sb` is declared as a required parameter in Fortran 95 interface for the function `sdot` to distinguish between function flavors that output final result in different precision.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

```
res = cdotc(n, x, incx, y, incy)
res = zdotc(n, x, incx, y, incy)
res = dotc(x, y)
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?dotc` routines perform a vector-vector operation defined as:

$$res = \sum_{i=1}^n \text{conjg}(x_i) * y_i,$$

where x_i and y_i are elements of vectors x and y .

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in vectors x and y .
<code>x</code>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.
<code>incx</code>	INTEGER. Specifies the increment for the elements of x .
<code>y</code>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.
<code>incy</code>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<code>res</code>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code> Contains the result of the dot product of the conjugated x and unconjugated y , if n is positive. Otherwise, it contains 0.
------------------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `dotc` interface are the following:

<code>x</code>	Holds the vector with the number of elements n .
<code>y</code>	Holds the vector with the number of elements n .

?dotu

Computes a vector-vector dot product.

Syntax

`res = cdotu(n, x, incx, y, incy)`

```
res = zdotu(n, x, incx, y, incy)
```

```
res = dotu(x, y)
```

Include Files

- mkl.fi, blas.f90

Description

The `?dotu` routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^n x_i * y_i$$

where x_i and y_i are elements of complex vectors x and y .

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors x and y .
<i>x</i>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x .
<i>y</i>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<i>res</i>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Contains the result of the dot product of x and y , if n is positive. Otherwise, it contains 0.
------------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `dotu` interface are the following:

<i>x</i>	Holds the vector with the number of elements n .
<i>y</i>	Holds the vector with the number of elements n .

?nrm2

Computes the Euclidean norm of a vector.

Syntax

```

res = snrm2(n, x, incx)
res = dnrm2(n, x, incx)
res = scnrm2(n, x, incx)
res = dznm2(n, x, incx)
res = nrm2(x)

```

Include Files

- `mkl.fi, blas.f90`

Description

The ?nrm2 routines perform a vector reduction operation defined as

```
res = ||x||,
```

where:

x is a vector,

res is a value containing the Euclidean norm of the elements of *x*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vector <i>x</i> .
<i>x</i>	REAL for snrm2 DOUBLE PRECISION for dnrm2 COMPLEX for scnrm2 DOUBLE COMPLEX for dznm2 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>res</i>	REAL for snrm2 DOUBLE PRECISION for dnrm2 REAL for scnrm2 DOUBLE PRECISION for dznm2 Contains the Euclidean norm of the vector <i>x</i> .
------------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `nrm2` interface are the following:

x Holds the vector with the number of elements n .

?rot

Performs rotation of points in the plane.

Syntax

```
call srot( $n$ ,  $x$ ,  $incx$ ,  $y$ ,  $incy$ ,  $c$ ,  $s$ )
call drot( $n$ ,  $x$ ,  $incx$ ,  $y$ ,  $incy$ ,  $c$ ,  $s$ )
call csrot( $n$ ,  $x$ ,  $incx$ ,  $y$ ,  $incy$ ,  $c$ ,  $s$ )
call zdrot( $n$ ,  $x$ ,  $incx$ ,  $y$ ,  $incy$ ,  $c$ ,  $s$ )
call rot( $x$ ,  $y$ ,  $c$ ,  $s$ )
```

Include Files

- mkl.fi, blas.f90

Description

Given two complex vectors x and y , each vector element of these vectors is replaced as follows:

$$x_i = c \cdot x_i + s \cdot y_i$$

$$y_i = c \cdot y_i - s \cdot x_i$$

Input Parameters

n	INTEGER. Specifies the number of elements in vectors x and y .
x	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, size at least $(1 + (n-1) \cdot \text{abs}(incx))$.
$incx$	INTEGER. Specifies the increment for the elements of x .
y	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, size at least $(1 + (n-1) \cdot \text{abs}(incy))$.
$incy$	INTEGER. Specifies the increment for the elements of y .
c	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot

A scalar.

s REAL for srot
DOUBLE PRECISION for drot
REAL for csrot
DOUBLE PRECISION for zdrot

A scalar.

Output Parameters

x Each element is replaced by $c*x + s*y$.

y Each element is replaced by $c*y - s*x$.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `rot` interface are the following:

x Holds the vector with the number of elements n .

y Holds the vector with the number of elements n .

?rotg

Computes the parameters for a Givens rotation.

Syntax

```
call srotg(a, b, c, s)
call drotg(a, b, c, s)
call crotg(a, b, c, s)
call zrotg(a, b, c, s)
call rotg(a, b, c, s)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given the Cartesian coordinates (a, b) of a point, these routines return the parameters c, s, r , and z associated with the Givens rotation. The parameters c and s define a unitary matrix such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The parameter z is defined such that if $|a| > |b|$, z is s ; otherwise if c is not 0 z is $1/c$; otherwise z is 1.

See a more accurate LAPACK version [?lartg](#).

Input Parameters

<i>a</i>	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Provides the x-coordinate of the point <i>p</i> .
<i>b</i>	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Provides the y-coordinate of the point <i>p</i> .

Output Parameters

<i>a</i>	Contains the parameter <i>r</i> associated with the Givens rotation.
<i>b</i>	Contains the parameter <i>z</i> associated with the Givens rotation.
<i>c</i>	REAL for srotg DOUBLE PRECISION for drotg REAL for crotg DOUBLE PRECISION for zrotg Contains the parameter <i>c</i> associated with the Givens rotation.
<i>s</i>	REAL for srotg DOUBLE PRECISION for drotg COMPLEX for crotg DOUBLE COMPLEX for zrotg Contains the parameter <i>s</i> associated with the Givens rotation.

?rotm

Performs modified Givens rotation of points in the plane.

Syntax

```
call srotm(n, x, incx, y, incy, param)  
call drotm(n, x, incx, y, incy, param)  
call rotm(x, y, param)
```

Include Files

- `mkl.fi`, `blas.f90`

Description

Given two vectors x and y , each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for $i=1$ to n , where H is a modified Givens transformation matrix whose values are stored in the $param(2)$ through $param(5)$ array. See discussion on the $param$ argument.

Input Parameters

n INTEGER. Specifies the number of elements in vectors x and y .

x REAL for `srotm`
DOUBLE PRECISION for `drotm`
Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.

$incx$ INTEGER. Specifies the increment for the elements of x .

y REAL for `srotm`
DOUBLE PRECISION for `drotm`
Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.

$incy$ INTEGER. Specifies the increment for the elements of y .

$param$ REAL for `srotm`
DOUBLE PRECISION for `drotm`
Array, size 5.

The elements of the $param$ array are:

$param(1)$ contains a switch, $flag$. $param(2-5)$ contain h_{11} , h_{21} , h_{12} , and h_{22} , respectively, the components of the array H .

Depending on the values of $flag$, the components of H are set as follows:

$$flag = -1.0: H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

$$flag = 0.0: H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

$$flag = 1.0: H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

$$flag = -2.0: H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of $flag$ and are not required to be set in the $param$ vector.

Output Parameters

x	Each element $x(i)$ is replaced by $h_{11} * x(i) + h_{12} * y(i)$.
y	Each element $y(i)$ is replaced by $h_{21} * x(i) + h_{22} * y(i)$.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `rotm` interface are the following:

x	Holds the vector with the number of elements n .
y	Holds the vector with the number of elements n .

?rotmg

Computes the parameters for a modified Givens rotation.

Syntax

```
call srotmg(d1, d2, x1, y1, param)
call drotmg(d1, d2, x1, y1, param)
call rotmg(d1, d2, x1, y1, param)
```

Include Files

- `mk1.fi, blas.f90`

Description

Given Cartesian coordinates $(x1, y1)$ of an input vector, these routines compute the components of a modified Givens transformation matrix H that zeros the y -component of the resulting vector:

$$\begin{bmatrix} x1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x1\sqrt{d1} \\ y1\sqrt{d2} \end{bmatrix}$$

Input Parameters

$d1$	REAL for <code>srotmg</code> DOUBLE PRECISION for <code>drotmg</code> Provides the scaling factor for the x -coordinate of the input vector.
$d2$	REAL for <code>srotmg</code> DOUBLE PRECISION for <code>drotmg</code> Provides the scaling factor for the y -coordinate of the input vector.
$x1$	REAL for <code>srotmg</code> DOUBLE PRECISION for <code>drotmg</code> Provides the x -coordinate of the input vector.

y1 REAL for srotmg
 DOUBLE PRECISION for drotmg
 Provides the *y*-coordinate of the input vector.

Output Parameters

d1 REAL for srotmg
 DOUBLE PRECISION for drotmg
 Provides the first diagonal element of the updated matrix.

d2 REAL for srotmg
 DOUBLE PRECISION for drotmg
 Provides the second diagonal element of the updated matrix.

x1 REAL for srotmg
 DOUBLE PRECISION for drotmg
 Provides the *x*-coordinate of the rotated vector before scaling.

param REAL for srotmg
 DOUBLE PRECISION for drotmg
 Array, size 5.
 The elements of the *param* array are:
param(1) contains a switch, *flag*. the other array elements *param*(2-5) contain the components of the array *H*: h_{11} , h_{21} , h_{12} , and h_{22} , respectively.
 Depending on the values of *flag*, the components of *H* are set as follows:

$$flag = -1.0: H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

$$flag = 0.0: H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

$$flag = 1.0: H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

$$flag = -2.0: H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of *flag* and are not required to be set in the *param* vector.

?scal

Computes the product of a vector by a scalar.

Syntax

call sscal(*n*, *a*, *x*, *incx*)

call dscal(*n*, *a*, *x*, *incx*)

```
call cscal(n, a, x, incx)
call zscal(n, a, x, incx)
call csscal(n, a, x, incx)
call zdscal(n, a, x, incx)
call scal(x, a)
```

Include Files

- mkl.fi, blas.f90

Description

The ?scal routines perform a vector operation defined as

```
x = a*x
```

where:

a is a scalar, *x* is an *n*-element vector.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vector <i>x</i> .
<i>a</i>	REAL for sscal and csscal DOUBLE PRECISION for dscal and zdscal COMPLEX for cscal DOUBLE COMPLEX for zscal Specifies the scalar <i>a</i> .
<i>x</i>	REAL for sscal DOUBLE PRECISION for dscal COMPLEX for cscal and csscal DOUBLE COMPLEX for zscal and zdscal Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>x</i>	Updated vector <i>x</i> .
----------	---------------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `scal` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
----------	---

?swap

Swaps a vector with another vector.

Syntax

```
call sswap(n, x, incx, y, incy)
call dswap(n, x, incx, y, incy)
call cswap(n, x, incx, y, incy)
call zswap(n, x, incx, y, incy)
call swap(x, y)
```

Include Files

- `mkl.fi`, `blas.f90`

Description

Given two vectors x and y , the ?swap routines return vectors y and x swapped, each replacing the other.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors x and y .
<i>x</i>	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x .
<i>y</i>	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least $(1 + (n-1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<i>x</i>	Contains the resultant vector x , that is, the input vector y .
<i>y</i>	Contains the resultant vector y , that is, the input vector x .

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `swap` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .

i?amax

Finds the index of the element with maximum absolute value.

Syntax

`index = isamax(n, x, incx)`

`index = idamax(n, x, incx)`

`index = icamax(n, x, incx)`

`index = izamax(n, x, incx)`

`index = iamax(x)`

Include Files

- `mkl.fi, blas.f90`

Description

Given a vector *x*, the *i?amax* functions return the position of the vector element $x(i)$ that has the largest absolute value for real flavors, or the largest sum $|\operatorname{Re}(x(i))| + |\operatorname{Im}(x(i))|$ for complex flavors.

If *n* is not positive, 0 is returned.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vector <i>x</i> .
<i>x</i>	REAL for <i>isamax</i> DOUBLE PRECISION for <i>idamax</i> COMPLEX for <i>icamax</i> DOUBLE COMPLEX for <i>izamax</i> Array, size at least $(1+(n-1)*\operatorname{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>index</i>	INTEGER. Contains the position of vector element that has the largest absolute value such that $x(index)$ has the largest absolute value.
--------------	---

BLAS 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the function *iamax* interface are the following:

x Holds the vector with the number of elements *n*.

i?amin

Finds the index of the element with the smallest absolute value.

Syntax

```
index = isamin(n, x, incx)
```

```
index = idamin(n, x, incx)
```

```
index = icamin(n, x, incx)
```

```
index = izamin(n, x, incx)
```

```
index = iamin(x)
```

Include Files

- `mkl.fi, blas.f90`

Description

Given a vector *x*, the *i?amin* functions return the position of the vector element $x(i)$ that has the smallest absolute value for real flavors, or the smallest sum $|\operatorname{Re}(x(i))| + |\operatorname{Im}(x(i))|$ for complex flavors.

If *n* is not positive, 0 is returned.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

Input Parameters

n INTEGER. On entry, *n* specifies the number of elements in vector *x*.

x REAL for *isamin*
 DOUBLE PRECISION for *idamin*
 COMPLEX for *icamin*
 DOUBLE COMPLEX for *izamin*
 Array, size at least $(1+(n-1)*\operatorname{abs}(incx))$.

incx INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

index INTEGER. Indicates the position of vector element with the smallest absolute value such that $x(index)$ has the smallest absolute value.

BLAS 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the function *iamin* interface are the following:

x Holds the vector with the number of elements *n*.

?cabs1

Computes absolute value of complex number.

Syntax

res = scabs1(*z*)

res = dcabs1(*z*)

res = cabs1(*z*)

Include Files

- mkl.fi, blas.f90

Description

The ?cabs1 is an auxiliary routine for a few BLAS Level 1 routines. This routine performs an operation defined as

$$res = |\operatorname{Re}(z)| + |\operatorname{Im}(z)|,$$

where *z* is a scalar, and *res* is a value containing the absolute value of a complex number *z*.

Input Parameters

z COMPLEX scalar for scabs1.
DOUBLE COMPLEX scalar for dcabs1.

Output Parameters

res REAL for scabs1.
DOUBLE PRECISION for dcabs1.
Contains the absolute value of a complex number *z*.

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. [Table "BLAS Level 2 Routine Groups and Their Data Types"](#) lists the BLAS Level 2 routine groups and the data types associated with them.

BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
?gbmv	s, d, c, z	Matrix-vector product using a general band matrix
?gemv	s, d, c, z	Matrix-vector product using a general matrix
?ger	s, d	Rank-1 update of a general matrix
?gerc	c, z	Rank-1 update of a conjugated general matrix
?geru	c, z	Rank-1 update of a general matrix, unconjugated
?hbmV	c, z	Matrix-vector product using a Hermitian band matrix

Routine Groups	Data Types	Description
?hemv	c, z	Matrix-vector product using a Hermitian matrix
?her	c, z	Rank-1 update of a Hermitian matrix
?her2	c, z	Rank-2 update of a Hermitian matrix
?hpmv	c, z	Matrix-vector product using a Hermitian packed matrix
?hpr	c, z	Rank-1 update of a Hermitian packed matrix
?hpr2	c, z	Rank-2 update of a Hermitian packed matrix
?sbmv	s, d	Matrix-vector product using symmetric band matrix
?spmv	s, d	Matrix-vector product using a symmetric packed matrix
?spr	s, d	Rank-1 update of a symmetric packed matrix
?spr2	s, d	Rank-2 update of a symmetric packed matrix
?symv	s, d	Matrix-vector product using a symmetric matrix
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Solution of a linear system of equations with a triangular band matrix
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Solution of a linear system of equations with a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trsv	s, d, c, z	Solution of a linear system of equations with a triangular matrix

?gbmv

Computes a matrix-vector product using a general band matrix

Syntax

```
call sgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call dgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call cgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call zgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call gbm(a, x, y [,kl] [,m] [,alpha] [,beta] [,trans])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gbmv routines perform a matrix-vector operation defined as

$y := \alpha * A * x + \beta * y,$

or

$y := \alpha * A' * x + \beta * y,$

or

$y := \alpha * \text{conjg}(A') * x + \beta * y,$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n band matrix, with kl sub-diagonals and ku super-diagonals.

Input Parameters

<i>trans</i>	<p>CHARACTER*1. Specifies the operation:</p> <p>If <i>trans</i>= 'N' or 'n', then $y := \alpha * A * x + \beta * y$</p> <p>If <i>trans</i>= 'T' or 't', then $y := \alpha * A' * x + \beta * y$</p> <p>If <i>trans</i>= 'C' or 'c', then $y := \alpha * \text{conjg}(A') * x + \beta * y$</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix A.</p> <p>The value of m must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix A.</p> <p>The value of n must be at least zero.</p>
<i>kl</i>	<p>INTEGER. Specifies the number of sub-diagonals of the matrix A.</p> <p>The value of kl must satisfy $0 \leq kl$.</p>
<i>ku</i>	<p>INTEGER. Specifies the number of super-diagonals of the matrix A.</p> <p>The value of ku must satisfy $0 \leq ku$.</p>
<i>alpha</i>	<p>REAL for <i>s</i>gbmv</p> <p>DOUBLE PRECISION for <i>d</i>gbmv</p> <p>COMPLEX for <i>c</i>gbmv</p> <p>DOUBLE COMPLEX for <i>z</i>gbmv</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <i>s</i>gbmv</p> <p>DOUBLE PRECISION for <i>d</i>gbmv</p> <p>COMPLEX for <i>c</i>gbmv</p> <p>DOUBLE COMPLEX for <i>z</i>gbmv</p> <p>Array, size (<i>lda</i>, <i>n</i>).</p> <p>Before entry, the leading $(kl + ku + 1)$ by n part of the array <i>a</i> must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(ku + 1)$ of the array, the first super-diagonal starting at position 2 in row ku, the first sub-</p>

diagonal starting at position 1 in row $(ku + 2)$, and so on. Elements in the array a that do not correspond to elements in the band matrix (such as the top left ku by ku triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  k = ku + 1 - j
  do 10, i = max(1, j-ku), min(m, j+kl)
    a(k+i, j) = matrix(i,j)
10  continue
20  continue

```

<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $(kl + ku + 1)$.
<i>x</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$ when <i>trans</i> = 'N' or 'n', and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>beta</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Specifies the scalar beta. When <i>beta</i> is equal to zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Array, size at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Updated vector <i>y</i> .
----------	---------------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gbmv` interface are the following:

<code>a</code>	Holds the array <code>a</code> of size $(kl+ku+1, n)$. Contains a banded matrix $m*n$ with <code>kl</code> lower diagonal and <code>ku</code> upper diagonal.
<code>x</code>	Holds the vector with the number of elements <code>rx</code> , where $rx = n$ if <code>trans = 'N'</code> , $rx = m$ otherwise.
<code>y</code>	Holds the vector with the number of elements <code>ry</code> , where $ry = m$ if <code>trans = 'N'</code> , $ry = n$ otherwise.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<code>kl</code>	If omitted, assumed $kl = ku$, that is, the number of lower diagonals equals the number of the upper diagonals.
<code>ku</code>	Restored as $ku = lda - kl - 1$, where <code>lda</code> is the leading dimension of matrix <code>A</code> .
<code>m</code>	If omitted, assumed $m = n$, that is, a square matrix.
<code>alpha</code>	The default value is 1.
<code>beta</code>	The default value is 0.

?gemv

Computes a matrix-vector product using a general matrix

Syntax

```
call sgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call cgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call zgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call scgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dzgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call gemv(a, x, y [,alpha][,beta] [,trans])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?gemv` routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

or

$$y := \alpha * A' * x + \beta * y,$$

or

$$y := \alpha * \text{conjg}(A') * x + \beta * y,$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n matrix.

Input Parameters

<i>trans</i>	<p>CHARACTER*1. Specifies the operation:</p> <p>if <i>trans</i>= 'N' or 'n', then $y := \alpha * A * x + \beta * y$;</p> <p>if <i>trans</i>= 'T' or 't', then $y := \alpha * A' * x + \beta * y$;</p> <p>if <i>trans</i>= 'C' or 'c', then $y := \alpha * \text{conjg}(A') * x + \beta * y$.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix A. The value of m must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix A. The value of n must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>sgemv</i></p> <p>DOUBLE PRECISION for <i>dgemv</i></p> <p>COMPLEX for <i>cgemv</i>, <i>scgemv</i></p> <p>DOUBLE COMPLEX for <i>zgemv</i>, <i>dzgemv</i></p> <p>Specifies the scalar α.</p>
<i>a</i>	<p>REAL for <i>sgemv</i>, <i>scgemv</i></p> <p>DOUBLE PRECISION for <i>dgemv</i>, <i>dzgemv</i></p> <p>COMPLEX for <i>cgemv</i></p> <p>DOUBLE COMPLEX for <i>zgemv</i></p> <p>Array, size (<i>lda</i>, <i>n</i>).</p> <p>Before entry, the leading m-by-n part of the array a must contain the matrix of coefficients.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program.</p> <p>The value of <i>lda</i> must be at least $\max(1, m)$.</p>
<i>x</i>	<p>REAL for <i>sgemv</i></p> <p>DOUBLE PRECISION for <i>dgemv</i></p> <p>COMPLEX for <i>cgemv</i>, <i>scgemv</i></p> <p>DOUBLE COMPLEX for <i>zgemv</i>, <i>dzgemv</i></p>

Array, size at least $(1+(n-1)*abs(incx))$ when *trans*= 'N' or 'n' and at least $(1+(m - 1)*abs(incx))$ otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.
The value of *incx* must not be zero.

beta REAL for *sgemv*
DOUBLE PRECISION for *dgemv*
COMPLEX for *cgemv*, *scgemv*
DOUBLE COMPLEX for *zgemv*, *dzgemv*
Specifies the scalar *beta*. When *beta* is set to zero, then *y* need not be set on input.

y REAL for *sgemv*
DOUBLE PRECISION for *dgemv*
COMPLEX for *cgemv*, *scgemv*
DOUBLE COMPLEX for *zgemv*, *dzgemv*
Array, size at least $(1+(m - 1)*abs(incy))$ when *trans*= 'N' or 'n' and at least $(1+(n - 1)*abs(incy))$ otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*.
The value of *incy* must not be zero.

Output Parameters

y Updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *gemv* interface are the following:

a Holds the matrix *A* of size (m,n) .

x Holds the vector with the number of elements *rx* where $rx = n$ if *trans* = 'N', $rx = m$ otherwise.

y Holds the vector with the number of elements *ry* where $ry = m$ if *trans* = 'N', $ry = n$ otherwise.

trans Must be 'N', 'C', or 'T'.
The default value is 'N'.

alpha The default value is 1.

beta The default value is 0.

?ger

Performs a rank-1 update of a general matrix.

Syntax

```
call sger(m, n, alpha, x, incx, y, incy, a, lda)
```

```
call dger(m, n, alpha, x, incx, y, incy, a, lda)
```

```
call ger(a, x, y [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?ger routines perform a matrix-vector operation defined as

$$A := \alpha x y^T + A,$$

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

A is an *m*-by-*n* general matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sger DOUBLE PRECISION for dger Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for sger DOUBLE PRECISION for dger Array, size at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>m</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for sger DOUBLE PRECISION for dger Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .

<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	REAL for <i>sger</i> DOUBLE PRECISION for <i>dger</i> Array, size (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the updated matrix.
----------	------------------------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *ger* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>x</i>	Holds the vector with the number of elements <i>m</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>alpha</i>	The default value is 1.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

```
call cgerc(m, n, alpha, x, incx, y, incy, a, lda)
call zgerc(m, n, alpha, x, incx, y, incy, a, lda)
call gerc(a, x, y [,alpha])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The *?gerc* routines perform a matrix-vector operation defined as

```
 $A := \alpha * x * \text{conjg}(y') + A,$ 
```

where:

alpha is a scalar,

x is an m -element vector,
 y is an n -element vector,
 A is an m -by- n matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix A . The value of m must be at least zero.
n	INTEGER. Specifies the number of columns of the matrix A . The value of n must be at least zero.
$alpha$	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Specifies the scalar $alpha$.
x	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Array, size at least $(1 + (m - 1) * abs(incx))$. Before entry, the incremented array x must contain the m -element vector x .
$incx$	INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.
y	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Array, size at least $(1 + (n - 1) * abs(incy))$. Before entry, the incremented array y must contain the n -element vector y .
$incy$	INTEGER. Specifies the increment for the elements of y . The value of $incy$ must not be zero.
a	COMPLEX for <code>cgerc</code> DOUBLE COMPLEX for <code>zgerc</code> Array, size (lda, n) . Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.
lda	INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, m)$.

Output Parameters

a Overwritten by the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gerc` interface are the following:

<code>a</code>	Holds the matrix A of size (m,n) .
<code>x</code>	Holds the vector with the number of elements m .
<code>y</code>	Holds the vector with the number of elements n .
<code>alpha</code>	The default value is 1.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

```
call cgeru(m, n, alpha, x, incx, y, incy, a, lda)
call zgeru(m, n, alpha, x, incx, y, incy, a, lda)
call geru(a, x, y [,alpha])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?geru` routines perform a matrix-vector operation defined as

$$A := \alpha x y^T + A,$$

where:

- `alpha` is a scalar,
- `x` is an m -element vector,
- `y` is an n -element vector,
- `A` is an m -by- n matrix.

Input Parameters

<code>m</code>	INTEGER. Specifies the number of rows of the matrix A . The value of m must be at least zero.
<code>n</code>	INTEGER. Specifies the number of columns of the matrix A . The value of n must be at least zero.
<code>alpha</code>	COMPLEX for <code>cgeru</code> DOUBLE COMPLEX for <code>zgeru</code> Specifies the scalar <code>alpha</code> .

<i>x</i>	COMPLEX for <code>cgeru</code> DOUBLE COMPLEX for <code>zgeru</code> Array, size at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>m</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <code>cgeru</code> DOUBLE COMPLEX for <code>zgeru</code> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for <code>cgeru</code> DOUBLE COMPLEX for <code>zgeru</code> Array, size (lda, n) . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the updated matrix.
----------	------------------------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `geru` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>x</i>	Holds the vector with the number of elements <i>m</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>alpha</i>	The default value is 1.

?hbm_v

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

```
call chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
```

```
call zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
```

```
call hbmv(a, x, y [,uplo][,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hbmv routines perform a matrix-vector operation defined as $y := \alpha A x + \beta y$,

where:

α and β are scalars,

x and y are n -element vectors,

A is an n -by- n Hermitian band matrix, with k super-diagonals.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian band matrix A is used:</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix A is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix A is used.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix A. The value of n must be at least zero.</p>
<i>k</i>	<p>INTEGER. For <i>uplo</i> = 'U' or 'u' Specifies the number of super-diagonals of the matrix A.</p> <p>For <i>uplo</i> = 'L' or 'l': Specifies the number of sub-diagonals of the matrix A.</p> <p>The value of k must satisfy $0 \leq k$.</p>
<i>alpha</i>	<p>COMPLEX for chbmv</p> <p>DOUBLE COMPLEX for zhbmv</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>COMPLEX for chbmv</p> <p>DOUBLE COMPLEX for zhbmv</p> <p>Array, size (<i>lda</i>, <i>n</i>).</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row k, and so on. The top left k by k triangle of the array a is not referenced.</p>

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  m = k + 1 - j
  do 10, i = max( 1, j - k ), j
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

Before entry with *uplo* = 'L' or 'l', the leading $(k + 1)$ by n part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min( n, j + k )
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $(k + 1)$.
<i>x</i>	COMPLEX for <i>chbmv</i> DOUBLE COMPLEX for <i>zhbmv</i> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for <i>chbmv</i> DOUBLE COMPLEX for <i>zhbmv</i> Specifies the scalar <i>beta</i> .
<i>y</i>	COMPLEX for <i>chbmv</i> DOUBLE COMPLEX for <i>zhbmv</i> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `hbmV` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(k+1,n)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

```
call chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call hemv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- `mkl.fi, blas.f90`

Description

The ?hemv routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *a* is used.

If *uplo* = 'U' or 'u', then the upper triangular of the array *a* is used.

If *uplo* = 'L' or 'l', then the low triangular of the array *a* is used.

<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero then <i>y</i> need not be set on input.
<i>y</i>	COMPLEX for chemv DOUBLE COMPLEX for zhemv Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `hemv` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>x</code>	Holds the vector with the number of elements n .
<code>y</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>alpha</code>	The default value is 1.
<code>beta</code>	The default value is 0.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

```
call cher(uplo, n, alpha, x, incx, a, lda)
call zher(uplo, n, alpha, x, incx, a, lda)
call her(a, x [,uplo] [, alpha])
```

Include Files

- `mkl.fi, blas.f90`

Description

The ?her routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(x') + A,$$

where:

`alpha` is a real scalar,

`x` is an n -element vector,

`A` is an n -by- n Hermitian matrix.

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <code>a</code> is used. If <code>uplo = 'U'</code> or <code>'u'</code> , then the upper triangular of the array <code>a</code> is used. If <code>uplo = 'L'</code> or <code>'l'</code> , then the low triangular of the array <code>a</code> is used.
<code>n</code>	INTEGER. Specifies the order of the matrix <code>A</code> . The value of <code>n</code> must be at least zero.
<code>alpha</code>	REAL for <code>cher</code>

	DOUBLE PRECISION for <i>zher</i> Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for <i>cher</i> DOUBLE COMPLEX for <i>zher</i> Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	COMPLEX for <i>cher</i> DOUBLE COMPLEX for <i>zher</i> Array, size (lda, n) . Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *her* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

```
call cher2(uplo, n, alpha, x, incx, y, incy, a, lda)
call zher2(uplo, n, alpha, x, incx, y, incy, a, lda)
call her2(a, x, y [,uplo][,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?her2 routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>COMPLEX for cher2</p> <p>DOUBLE COMPLEX for zher2</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>COMPLEX for cher2</p> <p>DOUBLE COMPLEX for zher2</p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>
<i>y</i>	<p>COMPLEX for cher2</p> <p>DOUBLE COMPLEX for zher2</p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.</p>

<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for <i>cher2</i> DOUBLE COMPLEX for <i>zher2</i> Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *her2* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

```
call chpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
call zhpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

```
call hpmv(ap, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hpmv routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>COMPLEX for chpmv</p> <p>DOUBLE COMPLEX for zhpmv</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>ap</i>	<p>COMPLEX for chpmv</p> <p>DOUBLE COMPLEX for zhpmv</p> <p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p> <p>The imaginary parts of the diagonal elements need not be set and are assumed to be zero.</p>
<i>x</i>	<p>COMPLEX for chpmv</p> <p>DOUBLE PRECISION COMPLEX for zhpmv</p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for <i>chpmv</i> DOUBLE COMPLEX for <i>zhpmv</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero then <i>y</i> need not be set on input.
<i>y</i>	COMPLEX for <i>chpmv</i> DOUBLE COMPLEX for <i>zhpmv</i> Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *hpmv* interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n*(n+1)/2)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?hpr

Performs a rank-1 update of a Hermitian packed matrix.

Syntax

```
call chpr(uplo, n, alpha, x, incx, ap)
call zhpr(uplo, n, alpha, x, incx, ap)
call hpr(ap, x [,uplo] [, alpha])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?hpr` routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(x') + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'L' or 'l', the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <code>chpr</code></p> <p>DOUBLE PRECISION for <code>zhpr</code></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>COMPLEX for <code>chpr</code></p> <p>DOUBLE COMPLEX for <code>zhpr</code></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>. <i>incx</i> must not be zero.</p>
<i>ap</i>	<p>COMPLEX for <code>chpr</code></p> <p>DOUBLE COMPLEX for <code>zhpr</code></p> <p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>A</i>_{1,1}, <i>ap</i>(2) and <i>ap</i>(3) contain <i>A</i>_{1,2} and <i>A</i>_{2,2} respectively, and so on.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>A</i>_{1,1}, <i>ap</i>(2) and <i>ap</i>(3) contain <i>A</i>_{2,1} and <i>A</i>_{3,1} respectively, and so on.</p>

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
 The imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `hpr` interface are the following:

ap Holds the array *ap* of size $(n*(n+1)/2)$.
x Holds the vector with the number of elements *n*.
uplo Must be 'U' or 'L'. The default value is 'U'.
alpha The default value is 1.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

```
call chpr2(uplo, n, alpha, x, incx, y, incy, ap)
call zhpr2(uplo, n, alpha, x, incx, y, incy, ap)
call hpr2(ap, x, y [,uplo][,alpha])
```

Include Files

- `mk1.fi`, `blas.f90`

Description

The `?hpr2` routines perform a matrix-vector operation defined as

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i></p> <p>Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>
<i>y</i>	<p>COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i>-element vector <i>y</i>.</p>
<i>incy</i>	<p>INTEGER. Specifies the increment for the elements of <i>y</i>.</p> <p>The value of <i>incy</i> must not be zero.</p>
<i>ap</i>	<p>COMPLEX for <i>chpr2</i> DOUBLE COMPLEX for <i>zhpr2</i></p> <p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p> <p>The imaginary parts of the diagonal elements need not be set and are assumed to be zero.</p>

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
 The imaginary parts of the diagonal elements need are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `hpr2` interface are the following:

ap Holds the array *ap* of size $(n*(n+1)/2)$.
x Holds the vector with the number of elements *n*.
y Holds the vector with the number of elements *n*.
uplo Must be 'U' or 'L'. The default value is 'U'.
alpha The default value is 1.

?sbmv

Computes a matrix-vector product using a symmetric band matrix.

Syntax

```
call ssbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call dsbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call sbmv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?sbmv` routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric band matrix, with *k* super-diagonals.

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix *A* is used:

if *uplo* = 'U' or 'u' - upper triangular part;

if *uplo* = 'L' or 'l' - low triangular part.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k INTEGER. Specifies the number of super-diagonals of the matrix *A*. The value of *k* must satisfy $0 \leq k$.

alpha REAL for *ssbmv*
DOUBLE PRECISION for *dsbmv*
Specifies the scalar *alpha*.

a REAL for *ssbmv*
DOUBLE PRECISION for *dsbmv*
Array, size (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  m = k + 1 - j
  do 10, i = max( 1, j - k ), j
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min( n, j + k )
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least (*k* + 1).

x REAL for *ssbmv*
DOUBLE PRECISION for *dsbmv*

	Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the vector x .
<i>incx</i>	INTEGER. Specifies the increment for the elements of x . The value of <i>incx</i> must not be zero.
<i>beta</i>	REAL for <i>ssbmv</i> DOUBLE PRECISION for <i>dsbmv</i> Specifies the scalar <i>beta</i> .
<i>y</i>	REAL for <i>ssbmv</i> DOUBLE PRECISION for <i>dsbmv</i> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the vector y .
<i>incy</i>	INTEGER. Specifies the increment for the elements of y . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector y .
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *sbmv* interface are the following:

<i>a</i>	Holds the array a of size $(k+1, n)$.
<i>x</i>	Holds the vector with the number of elements n .
<i>y</i>	Holds the vector with the number of elements n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?spmv

Computes a matrix-vector product using a symmetric packed matrix.

Syntax

```
call sspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call dspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call spmv(ap, x, y [,uplo][,alpha][,beta])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?spmv` routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <code>sspmv</code></p> <p>DOUBLE PRECISION for <code>dspmv</code></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>ap</i>	<p>REAL for <code>sspmv</code></p> <p>DOUBLE PRECISION for <code>dspmv</code></p> <p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on.</p>
<i>x</i>	<p>REAL for <code>sspmv</code></p> <p>DOUBLE PRECISION for <code>dspmv</code></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>

<i>beta</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for <code>sspmv</code> DOUBLE PRECISION for <code>dspmv</code> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `spmv` interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n*(n+1)/2)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

```
call sspr(uplo, n, alpha, x, incx, ap)
call dspr(uplo, n, alpha, x, incx, ap)
call spr(ap, x [,uplo] [, alpha])
```

Include Files

- `mk1.fi, blas.f90`

Description

The ?spr routines perform a matrix-vector operation defined as

```
a:= alpha*x*x'+ A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>sspr</i></p> <p>DOUBLE PRECISION for <i>dspr</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>REAL for <i>sspr</i></p> <p>DOUBLE PRECISION for <i>dspr</i></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>
<i>ap</i>	<p>REAL for <i>sspr</i></p> <p>DOUBLE PRECISION for <i>dspr</i></p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains $A_{1,1}$, <i>ap</i>(2) and <i>ap</i>(3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p>

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `'l'`, overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `spr` interface are the following:

<code>ap</code>	Holds the array <code>ap</code> of size $(n*(n+1)/2)$.
<code>x</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be <code>'U'</code> or <code>'L'</code> . The default value is <code>'U'</code> .
<code>alpha</code>	The default value is 1.

?spr2

Performs a rank-2 update of a symmetric packed matrix.

Syntax

```
call sspr2(uplo, n, alpha, x, incx, y, incy, ap)
call dspr2(uplo, n, alpha, x, incx, y, incy, ap)
call spr2(ap, x, y [,uplo][,alpha])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?spr2` routines perform a matrix-vector operation defined as

$$A := \alpha * x * y' + \alpha * y * x' + A,$$

where:

`alpha` is a scalar,

`x` and `y` are n -element vectors,

`A` is an n -by- n symmetric matrix, supplied in packed form.

Input Parameters

`uplo` CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix `A` is supplied in the packed array `ap`.

If `uplo = 'U'` or `'u'`, then the upper triangular part of the matrix `A` is supplied in the packed array `ap`.

If `uplo = 'L'` or `'l'`, then the low triangular part of the matrix `A` is supplied in the packed array `ap`.

<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <i>sspr2</i> DOUBLE PRECISION for <i>dspr2</i> Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for <i>sspr2</i> DOUBLE PRECISION for <i>dspr2</i> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for <i>sspr2</i> DOUBLE PRECISION for <i>dspr2</i> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>ap</i>	REAL for <i>sspr2</i> DOUBLE PRECISION for <i>dspr2</i> Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains $A_{1,1}$, <i>ap</i> (2) and <i>ap</i> (3) contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains $A_{1,1}$, <i>ap</i> (2) and <i>ap</i> (3) contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

Output Parameters

<i>ap</i>	With <i>uplo</i> = 'U' or 'u', overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.
-----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *spr2* interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n*(n+1)/2)$.
-----------	---

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

```
call ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call symv(a, x, y [,uplo][,alpha] [,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?symv routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssymv DOUBLE PRECISION for dsymv Array, size (<i>lda</i> , <i>n</i>).

Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix *A* and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix *A* and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $\max(1, n)$.

x REAL for *ssymv*
DOUBLE PRECISION for *dsymv*
Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.
The value of *incx* must not be zero.

beta REAL for *ssymv*
DOUBLE PRECISION for *dsymv*
Specifies the scalar *beta*.
When *beta* is supplied as zero, then *y* need not be set on input.

y REAL for *ssymv*
DOUBLE PRECISION for *dsymv*
Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*.
The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *symv* interface are the following:

a Holds the matrix *A* of size (n, n) .
x Holds the vector with the number of elements *n*.
y Holds the vector with the number of elements *n*.
uplo Must be 'U' or 'L'. The default value is 'U'.
alpha The default value is 1.

beta The default value is 0.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

```
call ssyr(uplo, n, alpha, x, incx, a, lda)
call dsyr(uplo, n, alpha, x, incx, a, lda)
call syr(a, x [,uplo] [, alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syr routines perform a matrix-vector operation defined as

$$A := \alpha * x * x' + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssyr DOUBLE PRECISION for dsyr Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for ssyr DOUBLE PRECISION for dsyr Array, size at least $(1 + (n-1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	REAL for ssyr DOUBLE PRECISION for dsyr

Array, size (lda, n) .

Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix A and the strictly lower triangular part of a is not referenced.

Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix A and the strictly upper triangular part of a is not referenced.

lda

INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, n)$.

Output Parameters

a

With $uplo = 'U'$ or $'u'$, the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix.

With $uplo = 'L'$ or $'l'$, the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `syr` interface are the following:

a

Holds the matrix A of size (n,n) .

x

Holds the vector with the number of elements n .

$uplo$

Must be $'U'$ or $'L'$. The default value is $'U'$.

$alpha$

The default value is 1.

?syr2

Performs a rank-2 update of symmetric matrix.

Syntax

```
call ssyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

```
call dsyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

```
call syr2(a, x, y [,uplo][,alpha])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?syr2` routines perform a matrix-vector operation defined as

$$A := \alpha x x^T + \alpha y y^T + A,$$

where:

α is a scalar,

x and y are n -element vectors,
 A is an n -by- n symmetric matrix.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the array a is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array a is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array a is used.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix A. The value of n must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>ssyr2</i></p> <p>DOUBLE PRECISION for <i>dsyr2</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>REAL for <i>ssyr2</i></p> <p>DOUBLE PRECISION for <i>dsyr2</i></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n-element vector x.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of x.</p> <p>The value of <i>incx</i> must not be zero.</p>
<i>y</i>	<p>REAL for <i>ssyr2</i></p> <p>DOUBLE PRECISION for <i>dsyr2</i></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array y must contain the n-element vector y.</p>
<i>incy</i>	<p>INTEGER. Specifies the increment for the elements of y. The value of <i>incy</i> must not be zero.</p>
<i>a</i>	<p>REAL for <i>ssyr2</i></p> <p>DOUBLE PRECISION for <i>dsyr2</i></p> <p>Array, size (lda, n).</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the leading n-by-n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the leading n-by-n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.</p>

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `syrr2` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>x</i>	Holds the vector <i>x</i> of length <i>n</i> .
<i>y</i>	Holds the vector <i>y</i> of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

```
call stbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call tbmv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?tbmv` routines perform one of the matrix-vector operations defined as

$x := A*x$, or $x := A'*x$, or $x := \text{conjg}(A')*x$,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with $(k + 1)$ diagonals.

Input Parameters

uplo CHARACTER*1. Specifies whether the matrix *A* is an upper or lower triangular matrix:

uplo = 'U' OR 'u'
 if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans CHARACTER*1. Specifies the operation:
 if *trans*= 'N' or 'n', then $x := A*x$;
 if *trans*= 'T' or 't', then $x := A'*x$;
 if *trans*= 'C' or 'c', then $x := \text{conjg}(A)*x$.

diag CHARACTER*1. Specifies whether the matrix *A* is unit triangular:
 if *uplo* = 'U' or 'u' then the matrix is unit triangular;
 if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

k INTEGER. On entry with *uplo* = 'U' or 'u' specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *a*.
 The value of *k* must satisfy $0 \leq k$.

a REAL for stbmv
 DOUBLE PRECISION for dtbmv
 COMPLEX for ctbmv
 DOUBLE COMPLEX for ztbmv
 Array, size (*lda*, *n*).
 Before entry with *uplo* = 'U' or 'u', the leading (*k* + 1) by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row (*k* + 1) of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

      do 20, j = 1, n
        m = k + 1 - j
        do 10, i = max( 1, j - k ), j
          a( m + i, j ) = matrix( i, j )
10       continue
20      continue

```

Before entry with *uplo* = 'L' or 'l', the leading (*k* + 1) by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not

referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min( n, j + k )
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

Note that when *uplo* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x REAL for stbmv
 DOUBLE PRECISION for dtbmv
 COMPLEX for ctbmv
 DOUBLE COMPLEX for ztbmv

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.
 The value of *incx* must not be zero.

Output Parameters

x Overwritten with the transformed vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `tbmv` interface are the following:

a Holds the array *a* of size $(k+1,n)$.

x Holds the vector with the number of elements *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N', 'C', or 'T'.
 The default value is 'N'.

diag Must be 'N' or 'U'. The default value is 'N'.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

```
call stbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call tbsv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?tbsv routines solve one of the following systems of equations:

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular band matrix, with $(k + 1)$ diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is an upper or lower triangular matrix: if <i>uplo</i> = 'U' or 'u' the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the system of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A')*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.
<i>k</i>	INTEGER. On entry with <i>uplo</i> = 'U' or 'u', k specifies the number of super-diagonals of the matrix A . On entry with <i>uplo</i> = 'L' or 'l', k specifies the number of sub-diagonals of the matrix A . The value of k must satisfy $0 \leq k$.
<i>a</i>	REAL for stbsv

DOUBLE PRECISION for dtbsv

COMPLEX for ctbsv

DOUBLE COMPLEX for ztbsv

Array, size (lda, n) .

Before entry with $uplo = 'U'$ or $'u'$, the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row k , and so on. The top left k by k triangle of the array a is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage ($matrix$) to band storage (a):

```

do 20, j = 1, n
  m = k + 1 - j
  do 10, i = max( 1, j - k ), j
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

Before entry with $uplo = 'L'$ or $'l'$, the leading $(k + 1)$ by n part of the array a must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array a is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage ($matrix$) to band storage (a):

```

do 20, j = 1, n
  m = 1 - j
  do 10, i = j, min( n, j + k )
    a( m + i, j ) = matrix( i, j )
10  continue
20  continue

```

When $diag = 'U'$ or $'u'$, the elements of the array a corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda

INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $(k + 1)$.

x

REAL for stbsv

DOUBLE PRECISION for dtbsv

COMPLEX for ctbsv

DOUBLE COMPLEX for ztbsv

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element right-hand side vector b .

$incx$

INTEGER. Specifies the increment for the elements of x .

The value of $incx$ must not be zero.

Output Parameters

x Overwritten with the solution vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `tbsv` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(k+1,n)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

```
call stpmv(uplo, trans, diag, n, ap, x, incx)
call dtpmv(uplo, trans, diag, n, ap, x, incx)
call ctpmv(uplo, trans, diag, n, ap, x, incx)
call ztpmv(uplo, trans, diag, n, ap, x, incx)
call tpmv(ap, x [,uplo] [, trans] [,diag])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?tpmv` routines perform one of the matrix-vector operations defined as

$x := A^*x$, or $x := A^T x$, or $x := \text{conjg}(A^T)x$,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: <i>uplo</i> = 'U' OR 'u'
-------------	--

	if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	<p>CHARACTER*1. Specifies the operation:</p> <p>if <i>trans</i> = 'N' or 'n', then $x := A*x$;</p> <p>if <i>trans</i> = 'T' or 't', then $x := A'*x$;</p> <p>if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A')*x$.</p>
<i>diag</i>	<p>CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular:</p> <p>if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular;</p> <p>if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.</p>
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>ap</i>	<p>REAL for <i>stpmv</i></p> <p>DOUBLE PRECISION for <i>dtpmv</i></p> <p>COMPLEX for <i>ctpmv</i></p> <p>DOUBLE COMPLEX for <i>ztpmv</i></p> <p>Array, size at least $((n*(n + 1))/2)$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(1,2) and <i>a</i>(2,2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular matrix packed sequentially, column-by-column, so that <i>ap</i>(1) contains <i>a</i>(1,1), <i>ap</i>(2) and <i>ap</i>(3) contain <i>a</i>(2,1) and <i>a</i>(3,1) respectively, and so on. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced, but are assumed to be unity.</p>
<i>x</i>	<p>REAL for <i>stpmv</i></p> <p>DOUBLE PRECISION for <i>dtpmv</i></p> <p>COMPLEX for <i>ctpmv</i></p> <p>DOUBLE COMPLEX for <i>ztpmv</i></p> <p>Array, size at least $(1 + (n - 1)*\text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>

Output Parameters

x Overwritten with the transformed vector *x*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `tpmv` interface are the following:

<code>ap</code>	Holds the array <code>ap</code> of size $(n*(n+1)/2)$.
<code>x</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<code>diag</code>	Must be 'N' or 'U'. The default value is 'N'.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

```
call stpsv(uplo, trans, diag, n, ap, x, incx)
call dtpsv(uplo, trans, diag, n, ap, x, incx)
call ctpsv(uplo, trans, diag, n, ap, x, incx)
call ztpsv(uplo, trans, diag, n, ap, x, incx)
call tpsv(ap, x [,uplo] [, trans] [,diag])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?tpsv` routines solve one of the following systems of equations

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: <code>uplo = 'U' OR 'u'</code> if <code>uplo = 'L' or 'l'</code> , then the matrix is low triangular.
<code>trans</code>	CHARACTER*1. Specifies the system of equations: if <code>trans= 'N' or 'n'</code> , then $A*x = b$; if <code>trans= 'T' or 't'</code> , then $A'*x = b$;

	if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A') * x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>ap</i>	REAL for <i>stpsv</i> DOUBLE PRECISION for <i>dtpsv</i> COMPLEX for <i>ctpsv</i> DOUBLE COMPLEX for <i>ztpsv</i> Array, size at least $((n * (n + 1)) / 2)$. Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the triangular matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the triangular matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on. When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced, but are assumed to be unity.
<i>x</i>	REAL for <i>stpsv</i> DOUBLE PRECISION for <i>dtpsv</i> COMPLEX for <i>ctpsv</i> DOUBLE COMPLEX for <i>ztpsv</i> Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element right-hand side vector <i>b</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the solution vector <i>x</i> .
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *tpsv* interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n * (n + 1)) / 2$.
-----------	---

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

```
call strmv(uplo, trans, diag, n, a, lda, x, incx)
call dtrmv(uplo, trans, diag, n, a, lda, x, incx)
call ctrmv(uplo, trans, diag, n, a, lda, x, incx)
call ztrmv(uplo, trans, diag, n, a, lda, x, incx)
call trmv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

$x := A*x$, or $x := A'*x$, or $x := \text{conjg}(A)*x$,

where:

x is an *n*-element vector,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: <i>uplo</i> = 'U' OR 'u' if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $x := A*x$; if <i>trans</i> = 'T' or 't', then $x := A'*x$; if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A)*x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.

<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>a</i>	<p>REAL for <code>strmv</code></p> <p>DOUBLE PRECISION for <code>dtrmv</code></p> <p>COMPLEX for <code>ctrmv</code></p> <p>DOUBLE COMPLEX for <code>ztrmv</code></p> <p>Array, size (<i>lda</i>, <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.</p>
<i>x</i>	<p>REAL for <code>strmv</code></p> <p>DOUBLE PRECISION for <code>dtrmv</code></p> <p>COMPLEX for <code>ctrmv</code></p> <p>DOUBLE COMPLEX for <code>ztrmv</code></p> <p>Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i>-element vector <i>x</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>

Output Parameters

<i>x</i>	Overwritten with the transformed vector <i>x</i> .
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `trmv` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

diag Must be 'N' or 'U'. The default value is 'N'.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

```
call strsv(uplo, trans, diag, n, a, lda, x, incx)
call dtrsv(uplo, trans, diag, n, a, lda, x, incx)
call ctrsv(uplo, trans, diag, n, a, lda, x, incx)
call ztrsv(uplo, trans, diag, n, a, lda, x, incx)
call trsv(a, x [,uplo] [, trans] [,diag])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsv routines solve one of the systems of equations:

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: <i>uplo</i> = 'U' OR 'u' if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the systems of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A')*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.
<i>a</i>	REAL for strsv

DOUBLE PRECISION for `dtrsv`

COMPLEX for `ctrsv`

DOUBLE COMPLEX for `ztrsv`

Array, size (lda, n) . Before entry with `uplo = 'U'` or `'u'`, the leading n -by- n upper triangular part of the array a must contain the upper triangular matrix and the strictly lower triangular part of a is not referenced. Before entry with `uplo = 'L'` or `'l'`, the leading n -by- n lower triangular part of the array a must contain the lower triangular matrix and the strictly upper triangular part of a is not referenced.

When `diag = 'U'` or `'u'`, the diagonal elements of a are not referenced either, but are assumed to be unity.

`lda` INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of `lda` must be at least $\max(1, n)$.

`x` REAL for `strsv`
 DOUBLE PRECISION for `dtrsv`
 COMPLEX for `ctrsv`
 DOUBLE COMPLEX for `ztrsv`

Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element right-hand side vector b .

`incx` INTEGER. Specifies the increment for the elements of x .
 The value of `incx` must not be zero.

Output Parameters

`x` Overwritten with the solution vector x .

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `trsv` interface are the following:

`a` Holds the matrix a of size (n,n) .

`x` Holds the vector with the number of elements n .

`uplo` Must be `'U'` or `'L'`. The default value is `'U'`.

`trans` Must be `'N'`, `'C'`, or `'T'`.
 The default value is `'N'`.

`diag` Must be `'N'` or `'U'`. The default value is `'N'`.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. Table “BLAS Level 3 Routine Groups and Their Data Types” lists the BLAS Level 3 routine groups and the data types associated with them.

BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
?gemm	s, d, c, z	Computes a matrix-matrix product with general matrices.
?hemm	c, z	Computes a matrix-matrix product where one input matrix is Hermitian.
?gemm_batch	s, d, c, z	Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.
?herk	c, z	Performs a Hermitian rank-k update.
?her2k	c, z	Performs a Hermitian rank-2k update.
?symm	s, d, c, z	Computes a matrix-matrix product where one input matrix is symmetric.
?syrk	s, d, c, z	Performs a symmetric rank-k update.
?syr2k	s, d, c, z	Performs a symmetric rank-2k update.
?trmm	s, d, c, z	Computes a matrix-matrix product where one input matrix is triangular.
?trsm	s, d, c, z	Solves a triangular matrix equation.

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time executing BLAS routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The BLAS functions are blocked where possible to restructure the code in a way that increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- The code is distributed across the processors to maximize parallelism.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?gemm

Computes a matrix-matrix product with general matrices.

Syntax

```
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call scgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dzgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemm(a, b, c [,transa][,transb] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

See also ?gemm3m, BLAS-like extension routines, that use matrix multiplication for similar matrix-matrix operations.

Input Parameters

<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A^T$; if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = A^H$.
<i>transb</i>	CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if <i>transb</i> = 'N' or 'n', then $\text{op}(B) = B$; if <i>transb</i> = 'T' or 't', then $\text{op}(B) = B^T$; if <i>transb</i> = 'C' or 'c', then $\text{op}(B) = B^H$.

<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix $op(A)$ and of the matrix <i>C</i>. The value of <i>m</i> must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix $op(B)$ and the number of columns of the matrix <i>C</i>.</p> <p>The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. Specifies the number of columns of the matrix $op(A)$ and the number of rows of the matrix $op(B)$.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>sgemm</i></p> <p>DOUBLE PRECISION for <i>dgemm</i></p> <p>COMPLEX for <i>cgemm</i>, <i>scgemm</i></p> <p>DOUBLE COMPLEX for <i>zgemm</i>, <i>dzgemm</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <i>sgemm</i>, <i>scgemm</i></p> <p>DOUBLE PRECISION for <i>dgemm</i>, <i>dzgemm</i></p> <p>COMPLEX for <i>cgemm</i></p> <p>DOUBLE COMPLEX for <i>zgemm</i></p> <p>Array, size <i>lda</i> by <i>ka</i>, where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program.</p> <p>When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>b</i>	<p>REAL for <i>sgemm</i></p> <p>DOUBLE PRECISION for <i>dgemm</i></p> <p>COMPLEX for <i>cgemm</i>, <i>scgemm</i></p> <p>DOUBLE COMPLEX for <i>zgemm</i>, <i>dzgemm</i></p> <p>Array, size <i>ldb</i> by <i>kb</i>, where <i>kb</i> is <i>n</i> when <i>transa</i> = 'N' or 'n', and is <i>k</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>n</i>-by-<i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.</p> <p>When <i>transb</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, k)$, otherwise <i>ldb</i> must be at least $\max(1, n)$.</p>
<i>beta</i>	<p>REAL for <i>sgemm</i></p>

DOUBLE PRECISION for `dgemm`
 COMPLEX for `cgemm`, `scgemm`
 DOUBLE COMPLEX for `zgemm`, `dzgemm`

Specifies the scalar *beta*.

When *beta* is equal to zero, then *c* need not be set on input.

c REAL for `sgemm`
 DOUBLE PRECISION for `dgemm`
 COMPLEX for `cgemm`, `scgemm`
 DOUBLE COMPLEX for `zgemm`, `dzgemm`
 Array, size *ldc* by *n*. Before entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, except when *beta* is equal to zero, in which case *c* need not be set on entry.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.
 The value of *ldc* must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the *m*-by-*n* matrix $(\alpha * \text{op}(A) * \text{op}(B) + \beta * C)$.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gemm` interface are the following:

a Holds the matrix *A* of size (*ma*,*ka*) where
 $ka = k$ if *transa*='N',
 $ka = m$ otherwise,
 $ma = m$ if *transa*='N',
 $ma = k$ otherwise.

b Holds the matrix *B* of size (*mb*,*kb*) where
 $kb = n$ if *transb* = 'N',
 $kb = k$ otherwise,
 $mb = k$ if *transb* = 'N',
 $mb = n$ otherwise.

c Holds the matrix *C* of size (*m*,*n*).

transa Must be 'N', 'C', or 'T'.
 The default value is 'N'.

<i>transb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?hemm

Computes a matrix-matrix product where one input matrix is Hermitian.

Syntax

```
call chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call hemm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?hemm routines compute a scalar-matrix-matrix product using a Hermitian matrix *A* and a general matrix *B* and add the result to a scalar-matrix product using a general matrix *C*. The operation is defined as

$$C := \alpha A * B + \beta * C$$

or

$$C := \alpha B * A + \beta * C,$$

where:

alpha and *beta* are scalars,

A is a Hermitian matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether the Hermitian matrix <i>A</i> appears on the left or right in the operation as follows: if <i>side</i> = 'L' or 'l', then $C := \alpha A * B + \beta * C$; if <i>side</i> = 'R' or 'r', then $C := \alpha B * A + \beta * C$.
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the Hermitian matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the Hermitian matrix <i>A</i> is used.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>C</i> .

	The value of m must be at least zero.
n	<p>INTEGER. Specifies the number of columns of the matrix C.</p> <p>The value of n must be at least zero.</p>
$alpha$	<p>COMPLEX for chemm</p> <p>DOUBLE COMPLEX for zhemm</p> <p>Specifies the scalar $alpha$.</p>
a	<p>COMPLEX for chemm</p> <p>DOUBLE COMPLEX for zhemm</p> <p>Array, size (lda, ka), where ka is m when $side = 'L'$ or $'l'$ and is n otherwise. Before entry with $side = 'L'$ or $'l'$, the m-by-m part of the array a must contain the Hermitian matrix, such that when $uplo = 'U'$ or $'u'$, the leading m-by-m upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced, and when $uplo = 'L'$ or $'l'$, the leading m-by-m lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced.</p> <p>Before entry with $side = 'R'$ or $'r'$, the n-by-n part of the array a must contain the Hermitian matrix, such that when $uplo = 'U'$ or $'u'$, the leading n-by-n upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced, and when $uplo = 'L'$ or $'l'$, the leading n-by-n lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.</p>
lda	<p>INTEGER. Specifies the leading dimension of a as declared in the calling (sub) program. When $side = 'L'$ or $'l'$ then lda must be at least $\max(1, m)$, otherwise lda must be at least $\max(1, n)$.</p>
b	<p>COMPLEX for chemm</p> <p>DOUBLE COMPLEX for zhemm</p> <p>Array, size ldb by n.</p> <p>The leading m-by-n part of the array b must contain the matrix B.</p>
ldb	<p>INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program. ldb must be at least $\max(1, m)$.</p>
$beta$	<p>COMPLEX for chemm</p> <p>DOUBLE COMPLEX for zhemm</p> <p>Specifies the scalar $beta$.</p> <p>When $beta$ is supplied as zero, then c need not be set on input.</p>
c	<p>COMPLEX for chemm</p> <p>DOUBLE COMPLEX for zhemm</p>

Array, size (c, n) . Before entry, the leading m -by- n part of the array c must contain the matrix C , except when β is zero, in which case c need not be set on entry.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. *ldc* must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the m -by- n updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `hemm` interface are the following:

a Holds the matrix A of size (k,k) where
 $k = m$ if *side* = 'L',
 $k = n$ otherwise.

b Holds the matrix B of size (m,n) .

c Holds the matrix C of size (m,n) .

side Must be 'L' or 'R'. The default value is 'L'.

uplo Must be 'U' or 'L'. The default value is 'U'.

alpha The default value is 1.

beta The default value is 0.

?gemm_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

```
call sgemm_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call dgemm_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call cgemm_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call zgemm_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call gemm_batch(a_array, b_array, c_array, group_size [,transa_array][,transb_array]
[,alpha_array][,beta_array])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemm_batch routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm routine counterparts, but the ?gemm_batch routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters.

The operation is defined as

```
idx = 1
for i = 1..group_count
  alpha and beta in alpha_array(i) and beta_array(i)
  for j = 1..group_size(i)
    A, B, and C matrix in a_array(idx), b_array(idx), and c_array(idx)
    C := alpha*op(A)*op(B) + beta*C,
    idx = idx + 1
  end for
end for
```

where:

op(X) is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

alpha and beta are scalar elements of alpha_array and beta_array,

A, B and C are matrices such that for m, n, and k which are elements of m_array, n_array, and k_array:

op(A) is an m-by-k matrix,

op(B) is a k-by-n matrix,

C is an m-by-n matrix.

A, B, and C represent matrices stored at addresses pointed to by a_array, b_array, and c_array, respectively. The number of entries in a_array, b_array, and c_array is total_batch_count = the sum of all of the group_size entries.

See also gemm for a detailed description of multiplication for general matrices and ?gemm3m_batch, BLAS-like extension routines for similar matrix-matrix operations.

NOTE

Error checking is not performed for Intel MKL Windows* single dynamic libraries for the ?gemm_batch routines.

Input Parameters

transa_array CHARACTER*1. Array of size *group_count*. For the group *i*, *transa_i* = *transa_array(i)* specifies the form of op(A) used in the matrix multiplication:

if *transa_i* = 'N' or 'n', then $\text{op}(A) = A$;

if *transa_i* = 'T' or 't', then $\text{op}(A) = A^T$;

if *transa_i* = 'C' or 'c', then $\text{op}(A) = A^H$.

<i>transb_array</i>	<p>CHARACTER*1. Array of size <i>group_count</i>. For the group <i>i</i>, <i>transb_i</i> = <i>transb_array(i)</i> specifies the form of $\text{op}(B_i)$ used in the matrix multiplication:</p> <p>if <i>transb_i</i> = 'N' or 'n', then $\text{op}(B) = B$;</p> <p>if <i>transb_i</i> = 'T' or 't', then $\text{op}(B) = B^T$;</p> <p>if <i>transb_i</i> = 'C' or 'c', then $\text{op}(B) = B^H$.</p>
<i>m_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>m_i</i> = <i>m_array(i)</i> specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix <i>C</i>.</p> <p>The value of each element of <i>m_array</i> must be at least zero.</p>
<i>n_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>n_i</i> = <i>n_array(i)</i> specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix <i>C</i>.</p> <p>The value of each element of <i>n_array</i> must be at least zero.</p>
<i>k_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>k_i</i> = <i>k_array(i)</i> specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$.</p> <p>The value of each element of <i>k_array</i> must be at least zero.</p>
<i>alpha_array</i>	<p>REAL for <i>sgemm_batch</i></p> <p>DOUBLE PRECISION for <i>dgemm_batch</i></p> <p>COMPLEX for <i>cgemm_batch</i>, <i>scgemm_batch</i></p> <p>DOUBLE COMPLEX for <i>zgemm_batch</i>, <i>dzgemm_batch</i></p> <p>Array of size <i>group_count</i>. For the group <i>i</i>, <i>alpha_array(i)</i> specifies the scalar <i>alpha_i</i>.</p>
<i>a_array</i>	<p>INTEGER*8 for Intel® 64 architecture</p> <p>INTEGER*4 for IA-32 architecture</p> <p>Array, size <i>total_batch_count</i>, of pointers to arrays used to store <i>A</i> matrices.</p>
<i>lda_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>lda_i</i> = <i>lda_array(i)</i> specifies the leading dimension of the array storing matrix <i>A</i> as declared in the calling (sub)program.</p> <p>When <i>transa_i</i> = 'N' or 'n', then <i>lda_i</i> must be at least $\max(1, m_i)$, otherwise <i>lda_i</i> must be at least $\max(1, k_i)$.</p>
<i>b_array</i>	<p>INTEGER*8 for Intel® 64 architecture</p> <p>INTEGER*4 for IA-32 architecture</p> <p>Array, size <i>total_batch_count</i>, of pointers to arrays used to store <i>B</i> matrices.</p>
<i>ldb_array</i>	<p>INTEGER.</p>

Array of size *group_count*. For the group *i*, $ldb_i = ldb_array(i)$ specifies the leading dimension of the array storing matrix *B* as declared in the calling (sub)program.

When $transb_i = 'N'$ or $'n'$, then ldb_i must be at least $\max(1, k_i)$, otherwise ldb_i must be at least $\max(1, n_i)$.

beta_array

REAL for *sgemm_batch*

DOUBLE PRECISION for *dgemm_batch*

COMPLEX for *cgemm_batch*, *scgemm_batch*

DOUBLE COMPLEX for *zgemm_batch*, *dzgemm_batch*

For the group *i*, *beta_array(i)* specifies the scalar *beta_i*.

When *beta_i* is equal to zero, then *C* matrices in group *i* need not be set on input.

c_array

INTEGER*8 for Intel® 64 architecture

INTEGER*4 for IA-32 architecture

Array, size *total_batch_count*, of pointers to arrays used to store *C* matrices.

ldc_array

INTEGER.

Array of size *group_count*. For the group *i*, $ldc_i = ldc_array(i)$ specifies the leading dimension of all arrays storing matrix *C* in group *i* as declared in the calling (sub)program.

ldc_i must be at least $\max(1, m_i)$.

group_count

INTEGER.

Specifies the number of groups. Must be at least 0.

group_size

INTEGER.

Array of size *group_count*. The element *group_size(i)* specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array

Overwritten by the m_i -by- n_i matrix $(\alpha_i * op(A) * op(B) + \beta_i * C)$ for group *i*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *gemm_batch* interface are the following:

a_array

Holds pointers to arrays containing matrices *A* of size (m_a, k_a) where

$k_a = k$ if $transa = 'N'$,

$k_a = m$ otherwise,

	$ma = m$ if $transa='N'$,
	$ma = k$ otherwise.
<code>b_array</code>	Holds pointers to arrays containing matrices B of size (mb, kb) where $kb = n$ if $transb_array = 'N'$, $kb = k$ otherwise, $mb = k$ if $transb_array = 'N'$, $mb = n$ otherwise.
<code>c_array</code>	Holds pointers to arrays containing matrices C of size (m, n) .
<code>m_array</code>	Array indicating number of rows of matrices $op(A)$ and C for each group.
<code>n_array</code>	Array indicating number of columns of matrices $op(B)$ and C for each group.
<code>k_array</code>	Array indicating number of rows of matrices $op(A)$ and number of columns of matrices $op(B)$ for each group.
<code>transa_array</code>	Array with each element set to one of 'N', 'C', or 'T'. The default values are 'N'.
<code>transb_array</code>	Array with each element set to one of 'N', 'C', or 'T'. The default values are 'N'.
<code>alpha_array</code>	Array of <i>alpha</i> values; the default value is 1.
<code>beta_array</code>	Array of <i>beta</i> values; the default value is 0.

?herk

Performs a Hermitian rank-k update.

Syntax

```
call cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call herk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The ?herk routines perform a rank-k matrix-matrix operation using a general matrix A and a Hermitian matrix C . The operation is defined as

$$C := \alpha * A * A^H + \beta * C,$$

or

$$C := \alpha * A^H * A + \beta * C,$$

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.</p>
<i>trans</i>	<p>CHARACTER*1. Specifies the operation:</p> <p>if <i>trans</i>= 'N' or 'n', then $C := \alpha * A * A^H + \beta * C$;</p> <p>if <i>trans</i>= 'C' or 'c', then $C := \alpha * A^H * A + \beta * C$.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. With <i>trans</i>= 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>A</i>, and with <i>trans</i>= 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>A</i>.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>cherk</i></p> <p>DOUBLE PRECISION for <i>zherk</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>COMPLEX for <i>cherk</i></p> <p>DOUBLE COMPLEX for <i>zherk</i></p> <p>Array, size (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i>= 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i>= 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i>= 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>beta</i>	<p>REAL for <i>cherk</i></p> <p>DOUBLE PRECISION for <i>zherk</i></p> <p>Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>COMPLEX for <i>cherk</i></p> <p>DOUBLE COMPLEX for <i>zherk</i></p> <p>Array, size <i>ldc</i> by <i>n</i>.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>c</i> is not referenced.</p>

Before entry with `uplo = 'L' or 'l'`, the leading n -by- n lower triangular part of the array `c` must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of `c` is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

`ldc` INTEGER. Specifies the leading dimension of `c` as declared in the calling (sub)program. The value of `ldc` must be at least $\max(1, n)$.

Output Parameters

`c` With `uplo = 'U' or 'u'`, the upper triangular part of the array `c` is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L' or 'l'`, the lower triangular part of the array `c` is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `herk` interface are the following:

<code>a</code>	Holds the matrix A of size (ma,ka) where $ka = k$ if <code>trans='N'</code> , $ka = n$ otherwise, $ma = n$ if <code>trans='N'</code> , $ma = k$ otherwise.
<code>c</code>	Holds the matrix C of size (n,n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.
<code>alpha</code>	The default value is 1.
<code>beta</code>	The default value is 0.

?her2k

Performs a Hermitian rank-2k update.

Syntax

```
call cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call her2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?her2k` routines perform a rank-2k matrix-matrix operation using general matrices A and B and a Hermitian matrix C . The operation is defined as

$$C := \alpha * A * B^H + \text{conjg}(\alpha) * B * A^H + \text{beta} * C,$$

or

$$C := \alpha * A^H * B + \text{conjg}(\alpha) * B^H * A + \text{beta} * C,$$

where:

α is a scalar and beta is a real scalar,

C is an n -by- n Hermitian matrix,

A and B are n -by- k matrices in the first case and k -by- n matrices in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * B^H + \alpha * B * A^H + \text{beta} * C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha * A^H * B + \alpha * B^H * A + \text{beta} * C$.
<i>n</i>	INTEGER. Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	INTEGER. With <i>trans</i> = 'N' or 'n' specifies the number of columns of the matrix A , and with <i>trans</i> = 'C' or 'c', k specifies the number of rows of the matrix A . The value of k must be at least equal to zero.
<i>alpha</i>	COMPLEX for <code>cher2k</code> DOUBLE COMPLEX for <code>zher2k</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <code>cher2k</code> DOUBLE COMPLEX for <code>zher2k</code> Array, size (<i>lda</i> , <i>ka</i>), where <i>ka</i> is k when <i>trans</i> = 'N' or 'n', and is n otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading n -by- k part of the array <i>a</i> must contain the matrix A , otherwise the leading k -by- n part of the array <i>a</i> must contain the matrix A .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program.

When $trans = 'N'$ or $'n'$, then lda must be at least $\max(1, n)$, otherwise lda must be at least $\max(1, k)$.

<i>beta</i>	<p>REAL for <code>cher2k</code></p> <p>DOUBLE PRECISION for <code>zher2k</code></p> <p>Specifies the scalar <i>beta</i>.</p>
<i>b</i>	<p>COMPLEX for <code>cher2k</code></p> <p>DOUBLE COMPLEX for <code>zher2k</code></p> <p>Array, size (ldb, kb), where kb is k when $trans = 'N'$ or $'n'$, and is n otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading n-by-k part of the array b must contain the matrix B, otherwise the leading k-by-n part of the array b must contain the matrix B.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program.</p> <p>When $trans = 'N'$ or $'n'$, then ldb must be at least $\max(1, n)$, otherwise ldb must be at least $\max(1, k)$.</p>
<i>c</i>	<p>COMPLEX for <code>cher2k</code></p> <p>DOUBLE COMPLEX for <code>zher2k</code></p> <p>Array, size ldc by n.</p> <p>Before entry with $uplo = 'U'$ or $'u'$, the leading n-by-n upper triangular part of the array c must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of c is not referenced.</p> <p>Before entry with $uplo = 'L'$ or $'l'$, the leading n-by-n lower triangular part of the array c must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of c is not referenced.</p> <p>The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.</p>
<i>ldc</i>	<p>INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, n)$.</p>

Output Parameters

<i>c</i>	<p>With $uplo = 'U'$ or $'u'$, the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.</p> <p>With $uplo = 'L'$ or $'l'$, the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.</p> <p>The imaginary parts of the diagonal elements are set to zero.</p>
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `her2k` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>ma,ka</i>) where $ka = k$ if <i>trans</i> = 'N', $ka = n$ otherwise, $ma = n$ if <i>trans</i> = 'N', $ma = k$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>mb,kb</i>) where $kb = k$ if <i>trans</i> = 'N', $kb = n$ otherwise, $mb = n$ if <i>trans</i> = 'N', $mb = k$ otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (<i>n,n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?symm

Computes a matrix-matrix product where one input matrix is symmetric.

Syntax

```
call ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call symm(a, b, c [,side][,uplo] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?symm routines compute a scalar-matrix-matrix product with one symmetric matrix and add the result to a scalar-matrix product. The operation is defined as

```
C := alpha*A*B + beta*C,
```

or

```
C := alpha*B*A + beta*C,
```

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and C are m -by- n matrices.

Input Parameters

<i>side</i>	<p>CHARACTER*1. Specifies whether the symmetric matrix A appears on the left or right in the operation:</p> <p>if <i>side</i> = 'L' or 'l', then $C := \alpha * A * B + \beta * C$;</p> <p>if <i>side</i> = 'R' or 'r', then $C := \alpha * B * A + \beta * C$.</p>
<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is used:</p> <p>if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used;</p> <p>if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix C.</p> <p>The value of m must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix C.</p> <p>The value of n must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>ssymm</i></p> <p>DOUBLE PRECISION for <i>dsymm</i></p> <p>COMPLEX for <i>csymm</i></p> <p>DOUBLE COMPLEX for <i>zsymm</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <i>ssymm</i></p> <p>DOUBLE PRECISION for <i>dsymm</i></p> <p>COMPLEX for <i>csymm</i></p> <p>DOUBLE COMPLEX for <i>zsymm</i></p> <p>Array, size (lda, ka), where ka is m when <i>side</i> = 'L' or 'l' and is n otherwise.</p> <p>Before entry with <i>side</i> = 'L' or 'l', the m-by-m part of the array a must contain the symmetric matrix, such that when <i>uplo</i> = 'U' or 'u', the leading m-by-m upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced, and when <i>side</i> = 'L' or 'l', the leading m-by-m lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.</p> <p>Before entry with <i>side</i> = 'R' or 'r', the n-by-n part of the array a must contain the symmetric matrix, such that when <i>uplo</i> = 'U' or 'u' the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced, and when <i>side</i> = 'L' or 'l', the leading n-by-n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.</p>

<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l' then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, n)$.</p>
<i>b</i>	<p>REAL for <i>ssymm</i> DOUBLE PRECISION for <i>dsymm</i> COMPLEX for <i>csymm</i> DOUBLE COMPLEX for <i>zsymm</i></p> <p>Array, size <i>ldb</i> by <i>n</i>.</p> <p>The leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. <i>ldb</i> must be at least $\max(1, m)$.</p>
<i>beta</i>	<p>REAL for <i>ssymm</i> DOUBLE PRECISION for <i>dsymm</i> COMPLEX for <i>csymm</i> DOUBLE COMPLEX for <i>zsymm</i></p> <p>Specifies the scalar <i>beta</i>.</p> <p>When <i>beta</i> is set to zero, then <i>c</i> need not be set on input.</p>
<i>c</i>	<p>REAL for <i>ssymm</i> DOUBLE PRECISION for <i>dsymm</i> COMPLEX for <i>csymm</i> DOUBLE COMPLEX for <i>zsymm</i></p> <p>Array, size (<i>c</i>, <i>n</i>). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.</p>
<i>ldc</i>	<p>INTEGER. Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program. <i>ldc</i> must be at least $\max(1, m)$.</p>

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *symm* interface are the following:

<i>a</i>	<p>Holds the matrix <i>A</i> of size (<i>k</i>,<i>k</i>) where</p> <p>$k = m$ if <i>side</i> = 'L',</p> <p>$k = n$ otherwise.</p>
----------	---

<i>b</i>	Holds the matrix <i>B</i> of size (m,n) .
<i>c</i>	Holds the matrix <i>C</i> of size (m,n) .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?syrk

Performs a symmetric rank-k update.

Syntax

```
call ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zsykrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call syrk(a, c [,uplo] [, trans] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syrk routines perform a rank-k matrix-matrix operation for a symmetric matrix *C* using a general matrix *A*. The operation is defined as

$$C := \alpha A A' + \beta C,$$

or

$$C := \alpha A' A + \beta C,$$

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha A A' + \beta C$; if <i>trans</i> = 'T' or 't', then $C := \alpha A' A + \beta C$;

if $trans = 'C'$ or $'c'$, then $C := \alpha * A' * A + \beta * C$.

<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. On entry with $trans = 'N'$ or $'n'$, <i>k</i> specifies the number of columns of the matrix <i>a</i>, and on entry with $trans = 'T'$, $'t'$, $'C'$, or $'c'$, <i>k</i> specifies the number of rows of the matrix <i>a</i>.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <code>ssyrk</code> DOUBLE PRECISION for <code>dsyrk</code> COMPLEX for <code>csyrk</code> DOUBLE COMPLEX for <code>zsyrk</code></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <code>ssyrk</code> DOUBLE PRECISION for <code>dsyrk</code> COMPLEX for <code>csyrk</code> DOUBLE COMPLEX for <code>zsyrk</code></p> <p>Array, size (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when $trans = 'N'$ or $'n'$, and is <i>n</i> otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p> <p>Array, size (<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when $trans = 'N'$ or $'n'$, and is <i>n</i> otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When $trans = 'N'$ or $'n'$, then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>beta</i>	<p>REAL for <code>ssyrk</code> DOUBLE PRECISION for <code>dsyrk</code> COMPLEX for <code>csyrk</code> DOUBLE COMPLEX for <code>zsyrk</code></p> <p>Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>REAL for <code>ssyrk</code> DOUBLE PRECISION for <code>dsyrk</code> COMPLEX for <code>csyrk</code> DOUBLE COMPLEX for <code>zsyrk</code></p>

Array, size (ldc, n) . Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of c is not referenced.

Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of c is not referenced.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c With $uplo = 'U'$ or $'u'$, the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.
With $uplo = 'L'$ or $'l'$, the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `syrk` interface are the following:

a Holds the matrix A of size (ma,ka) where
 $ka = k$ if $transa='N'$,
 $ka = n$ otherwise,
 $ma = n$ if $transa='N'$,
 $ma = k$ otherwise.

c Holds the matrix C of size (n,n) .

uplo Must be $'U'$ or $'L'$. The default value is $'U'$.

trans Must be $'N'$, $'C'$, or $'T'$.
The default value is $'N'$.

alpha The default value is 1.

beta The default value is 0.

?syr2k

Performs a symmetric rank-2k update.

Syntax

```
call ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
call zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call syr2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation for a symmetric matrix C using general matrices A and B . The operation is defined as

$$C := \alpha A B' + \alpha B A' + \beta C,$$

or

$$C := \alpha A' B + \alpha B' A + \beta C,$$

where:

α and β are scalars,

C is an n -by- n symmetric matrix,

A and B are n -by- k matrices in the first case, and k -by- n matrices in the second case.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha A B' + \alpha B A' + \beta C$; if <i>trans</i> = 'T' or 't', then $C := \alpha A' B + \alpha B' A + \beta C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha A' B + \alpha B' A + \beta C$.
<i>n</i>	INTEGER. Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', k specifies the number of columns of the matrices A and B , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', k specifies the number of rows of the matrices A and B . The value of k must be at least zero.
<i>alpha</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyr2k Specifies the scalar α .
<i>a</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (lda, ka) , where ka is k when $trans = 'N'$ or $'n'$, and is n otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading n -by- k part of the array a must contain the matrix A , otherwise the leading k -by- n part of the array a must contain the matrix A .

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program.

When $trans = 'N'$ or $'n'$, then lda must be at least $\max(1, n)$, otherwise lda must be at least $\max(1, k)$.

b REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (ldb, kb) , where kb is k when $trans = 'N'$ or $'n'$, and is n otherwise. Before entry with $trans = 'N'$ or $'n'$, the leading n -by- k part of the array b must contain the matrix B , otherwise the leading k -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program.

When $trans = 'N'$ or $'n'$, then ldb must be at least $\max(1, n)$, otherwise ldb must be at least $\max(1, k)$.

beta REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Specifies the scalar β .

c REAL for ssyr2k

DOUBLE PRECISION for dsyr2k

COMPLEX for csyr2k

DOUBLE COMPLEX for zsyr2k

Array, size (ldc, n) . Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of c is not referenced.

Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of c is not referenced.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, n)$.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `syrr2k` interface are the following:

a Holds the matrix *A* of size (ma,ka) where
 $ka = k$ if *trans* = 'N',
 $ka = n$ otherwise,
 $ma = n$ if *trans* = 'N',
 $ma = k$ otherwise.

b Holds the matrix *B* of size (mb,kb) where
 $kb = k$ if *trans* = 'N',
 $kb = n$ otherwise,
 $mb = n$ if *trans* = 'N',
 $mb = k$ otherwise.

c Holds the matrix *C* of size (n,n) .

uplo Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N', 'C', or 'T'.
 The default value is 'N'.

alpha The default value is 1.

beta The default value is 0.

?trmm

Computes a matrix-matrix product where one input matrix is triangular.

Syntax

```
call strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call trmm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?trmm` routines compute a scalar-matrix-matrix product with one triangular matrix. The operation is defined as

```
B := alpha*op(A)*B
```

or

```
B := alpha*B*op(A)
```

where:

alpha is a scalar,

B is an *m*-by-*n* matrix,

A is a unit, or non-unit, upper or lower triangular matrix

`op(A)` is one of `op(A) = A`, or `op(A) = A'`, or `op(A) = conjg(A')`.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether <code>op(A)</code> appears on the left or right of <i>B</i> in the operation: if <i>side</i> = 'L' or 'l', then <code>B := alpha*op(A)*B</code> ; if <i>side</i> = 'R' or 'r', then <code>B := alpha*B*op(A)</code> .
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: <i>uplo</i> = 'U' or 'u' if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	CHARACTER*1. Specifies the form of <code>op(A)</code> used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then <code>op(A) = A</code> ; if <i>transa</i> = 'T' or 't', then <code>op(A) = A'</code> ; if <i>transa</i> = 'C' or 'c', then <code>op(A) = conjg(A')</code> .
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	INTEGER. Specifies the number of rows of <i>B</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <code>strmm</code> DOUBLE PRECISION for <code>dtrmm</code> COMPLEX for <code>ctrmm</code>

DOUBLE COMPLEX for `ztrmm`

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a REAL for `strmm`

DOUBLE PRECISION for `dtrmm`

COMPLEX for `ctrmm`

DOUBLE COMPLEX for `ztrmm`

Array, size *lda* by *k*, where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *k* by *k* lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least $\max(1, m)$, when *side* = 'R' or 'r', then *lda* must be at least $\max(1, n)$.

b REAL for `strmm`

DOUBLE PRECISION for `dtrmm`

COMPLEX for `ctrmm`

DOUBLE COMPLEX for `ztrmm`

Array, size *ldb* by *n*. Before entry, the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. *ldb* must be at least $\max(1, m)$.

Output Parameters

b Overwritten by the transformed matrix.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `trmm` interface are the following:

a Holds the matrix *A* of size (*k*,*k*) where
 $k = m$ if *side* = 'L',
 $k = n$ otherwise.

<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>alpha</i>	The default value is 1.

?trsm

Solves a triangular matrix equation.

Syntax

```
call strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call trsm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

Include Files

- mkl.fi, blas.f90

Description

The ?trsm routines solve one of the following matrix equations:

$op(A)*X = alpha*B,$

or

$X*op(A) = alpha*B,$

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix

op(A) is one of $op(A) = A$, or $op(A) = A'$, or $op(A) = conjg(A')$.

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

side CHARACTER*1. Specifies whether *op(A)* appears on the left or right of *X* in the equation:

if *side* = 'L' or 'l', then $op(A)*X = alpha*B;$

if *side* = 'R' or 'r', then $X*op(A) = alpha*B.$

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular:</p> <p><i>uplo</i> = 'U' or 'u'</p> <p>if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.</p>
<i>transa</i>	<p>CHARACTER*1. Specifies the form of $op(A)$ used in the matrix multiplication:</p> <p>if <i>transa</i> = 'N' or 'n', then $op(A) = A$;</p> <p>if <i>transa</i> = 'T' or 't';</p> <p>if <i>transa</i> = 'C' or 'c', then $op(A) = conjg(A)$.</p>
<i>diag</i>	<p>CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular:</p> <p>if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular;</p> <p>if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of <i>B</i>. The value of <i>m</i> must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of <i>B</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>strsm</i></p> <p>DOUBLE PRECISION for <i>dtrsm</i></p> <p>COMPLEX for <i>ctrsm</i></p> <p>DOUBLE COMPLEX for <i>ztrsm</i></p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.</p>
<i>a</i>	<p>REAL for <i>strsm</i></p> <p>DOUBLE PRECISION for <i>dtrsm</i></p> <p>COMPLEX for <i>ctrsm</i></p> <p>DOUBLE COMPLEX for <i>ztrsm</i></p> <p>Array, size (<i>lda</i>, <i>k</i>), where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l' lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l', then <i>lda</i> must be at least $\max(1, m)$, when <i>side</i> = 'R' or 'r', then <i>lda</i> must be at least $\max(1, n)$.</p>

<i>b</i>	REAL for <code>strsm</code> DOUBLE PRECISION for <code>dtrsm</code> COMPLEX for <code>ctrsm</code> DOUBLE COMPLEX for <code>ztrsm</code> Array, size <i>ldb</i> by <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. <i>ldb</i> must be at least $\max(1, m)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `trsm` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>k</i>) where $k = m$ if <i>side</i> = 'L', $k = n$ otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>alpha</i>	The default value is 1.

Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® Math Kernel Library beginning with the Intel MKL release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If *nz* is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments

Compressed sparse vectors. Let a be a vector stored in an array, and assume that the only non-zero elements of a are the following:

$$a(k_1), a(k_2), a(k_3) \dots a(k_{nz}),$$

where nz is the total number of non-zero elements in a .

In Sparse BLAS, this vector can be represented in compressed form by two arrays, x (values) and $indx$ (indices). Each array has nz elements:

$$x(1)=a(k_1), x(2)=a(k_2), \dots x(nz)= a(k_{nz}),$$

$$indx(1)=k_1, indx(2)=k_2, \dots indx(nz)= k_{nz}.$$

Thus, a sparse vector is fully determined by the triple $(nz, x, indx)$. If you pass a negative or zero value of nz to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single array (a full-storage vector). If y is a full-storage vector, its elements must be stored contiguously: the first element in $y(1)$, the second in $y(2)$, and so on. This corresponds to an increment $incy = 1$ in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: s and d for single- and double-precision real; c and z for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix i (standing for *indexed*) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel MKL implementation of Sparse BLAS are listed in [Table "Sparse BLAS Routines and Their Data Types"](#).

Sparse BLAS Routines and Their Data Types

Routine/ Function	Data Types	Description
?axpyi	s, d, c, z	Scalar-vector product plus vector (routines)
?doti	s, d	Dot product (functions)
?dotci	c, z	Complex dot product conjugated (functions)
?dotui	c, z	Complex dot product unconjugated (functions)
?gthr	s, d, c, z	Gathering a full-storage sparse vector into compressed form $nz, x, indx$ (routines)
?gthrz	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
?roti	s, d	Givens rotation (routines)

Routine/ Function	Data Types	Description
<code>?sctr</code>	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array x (with the increment $incx=1$):

<code>?asum</code>	sum of absolute values of vector elements
<code>?copy</code>	copying a vector
<code>?nrm2</code>	Euclidean norm of a vector
<code>?scal</code>	scaling a vector
<code>i?amax</code>	index of the element with the largest absolute value for real flavors, or the largest sum $ \operatorname{Re}(x(i)) + \operatorname{Im}(x(i)) $ for complex flavors.
<code>i?amin</code>	index of the element with the smallest absolute value for real flavors, or the smallest sum $ \operatorname{Re}(x(i)) + \operatorname{Im}(x(i)) $ for complex flavors.

The result i returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is $x(i)$; the corresponding index in full-storage array is $indx(i)$.

You can also call `?rotg` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `?roti`.

`?axpyi`

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

```
call saxpyi(nz, a, x, indx, y)
call daxpyi(nz, a, x, indx, y)
call caxpyi(nz, a, x, indx, y)
call zaxpyi(nz, a, x, indx, y)
call axpyi(x, indx, y [, a])
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?axpyi` routines perform a vector-vector operation defined as

```
 $y := a*x + y$ 
```

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The `?axpyi` routines reference or modify only the elements of y whose indices are listed in the array $indx$.

The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
a	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Specifies the scalar a .
x	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Array, size at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Array, size at least $\max(indx(i))$.

Output Parameters

y	Contains the updated vector y .
-----	-----------------------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `axpyi` interface are the following:

x	Holds the vector with the number of elements nz .
$indx$	Holds the vector with the number of elements nz .
y	Holds the vector with the number of elements nz .
a	The default value is 1.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

```
res = sdoti(nz, x, indx, y )
```

```
res = ddoti(nz, x, indx, y )
```

```
res = doti(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?doti routines return the dot product of x and y defined as

```
res = x(1)*y(indx(1)) + x(2)*y(indx(2)) + ... + x(nz)*y(indx(nz))
```

where the triple $(nz, x, indx)$ defines a sparse real vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
x	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least $\max(indx(i))$.

Output Parameters

res	REAL for sdoti DOUBLE PRECISION for ddoti Contains the dot product of x and y , if nz is positive. Otherwise, res contains 0.
-------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `doti` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>nz</i> .
<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

```
res = cdotci(nz, x, indx, y )
```

```
res = zdotci(nzz, x, indx, y )
```

```
res = dotci(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?dotci routines return the dot product of *x* and *y* defined as

```
conjg(x(1))*y(indx(1)) + ... + conjg(x(nz))*y(indx(nz))
```

where the triple (*nz*, *x*, *indx*) defines a sparse complex vector stored in compressed form, and *y* is a real vector in full storage form. The functions reference only the elements of *y* whose indices are listed in the array *indx*. The values in *indx* must be distinct.

Input Parameters

<i>nz</i>	INTEGER. The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, size at least <i>nz</i> .
<i>indx</i>	INTEGER. Specifies the indices for the elements of <i>x</i> . Array, size at least <i>nz</i> .
<i>y</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci Array, size at least $\max(\text{indx}(i))$.

Output Parameters

<i>res</i>	COMPLEX for cdotci DOUBLE COMPLEX for zdotci
------------	---

Contains the conjugated dot product of x and y , if nz is positive. Otherwise, it contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `dotci` interface are the following:

x	Holds the vector with the number of elements (nz).
$indx$	Holds the vector with the number of elements (nz).
y	Holds the vector with the number of elements (nz).

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

```
res = cdotui(nz, x, indx, y)
res = zdotui(nzz, x, indx, y)
res = dotui(x, indx, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?dotui` routines return the dot product of x and y defined as

$$res = x(1)*y(indx(1)) + x(2)*y(indx(2)) + \dots + x(nz)*y(indx(nz))$$

where the triple $(nz, x, indx)$ defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
x	COMPLEX for <code>cdotui</code> DOUBLE COMPLEX for <code>zdotui</code> Array, size at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	COMPLEX for <code>cdotui</code> DOUBLE COMPLEX for <code>zdotui</code>

Array, size at least $\max(\text{indx}(i))$.

Output Parameters

res COMPLEX for `cdotui`
 DOUBLE COMPLEX for `zdotui`
 Contains the dot product of *x* and *y*, if *nz* is positive. Otherwise, *res* contains 0.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `dotui` interface are the following:

x Holds the vector with the number of elements *nz*.
indx Holds the vector with the number of elements *nz*.
y Holds the vector with the number of elements *nz*.

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

```
call sgthr(nz, y, x, indx )
call dgthr(nz, y, x, indx )
call cgthr(nz, y, x, indx )
call zgthr(nz, y, x, indx )
res = gthr(x, indx, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?gthr` routines gather the specified elements of a full-storage sparse vector *y* into compressed form(*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

$x(i) = y(\text{indx}(i))$, for $i=1, 2, \dots, nz$.

Input Parameters

nz INTEGER. The number of elements of *y* to be gathered.
indx INTEGER. Specifies indices of elements to be gathered.
 Array, size at least *nz*.

y REAL for `sgthr`
 DOUBLE PRECISION for `dgthr`
 COMPLEX for `cgthr`
 DOUBLE COMPLEX for `zgthr`
 Array, size at least $\max(\text{indx}(i))$.

Output Parameters

x REAL for `sgthr`
 DOUBLE PRECISION for `dgthr`
 COMPLEX for `cgthr`
 DOUBLE COMPLEX for `zgthr`
 Array, size at least *nz*.
 Contains the vector converted to the compressed form.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gthr` interface are the following:

x Holds the vector with the number of elements *nz*.
indx Holds the vector with the number of elements *nz*.
y Holds the vector with the number of elements *nz*.

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

```
call sgthrz(nz, y, x, indx )
call dgthrz(nz, y, x, indx )
call cgthrz(nz, y, x, indx )
call zgthrz(nz, y, x, indx )
res = gthrz(x, indx, y)
```

Include Files

- `mkl.fi, blas.f90`

Description

The `?gthrz` routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also `?gthr`).

Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>y</i> to be gathered.
<i>indx</i>	INTEGER. Specifies indices of elements to be gathered. Array, size at least <i>nz</i> .
<i>y</i>	REAL for <i>sgthrz</i> DOUBLE PRECISION for <i>dgthrz</i> COMPLEX for <i>cgthrz</i> DOUBLE COMPLEX for <i>zgthrz</i> Array, size at least $\max(\text{indx}(i))$.

Output Parameters

<i>x</i>	REAL for <i>sgthrz</i> DOUBLE PRECISION for <i>dgthrz</i> COMPLEX for <i>cgthrz</i> DOUBLE COMPLEX for <i>zgthrz</i> Array, size at least <i>nz</i> . Contains the vector converted to the compressed form.
<i>y</i>	The updated vector <i>y</i> .

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *gthrz* interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>nz</i> .
<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

```
call sroti(nz, x, indx, y, c, s)
call droti(nz, x, indx, y, c, s)
call roti(x, indx, y, c, s)
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?roti` routines apply the Givens rotation to elements of two real vectors, x (in compressed form nz , x , $indx$) and y (in full storage form):

$$x(i) = c*x(i) + s*y(indx(i))$$

$$y(indx(i)) = c*y(indx(i)) - s*x(i)$$

The routines reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
x	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> Array, size at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> Array, size at least $\max(indx(i))$.
c	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> . A scalar.
s	REAL for <code>sroti</code> DOUBLE PRECISION for <code>droti</code> . A scalar.

Output Parameters

x and y	The updated arrays.
-------------	---------------------

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `roti` interface are the following:

x	Holds the vector with the number of elements nz .
$indx$	Holds the vector with the number of elements nz .

y Holds the vector with the number of elements *nz*.

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

```
call ssctr(nz, x, indx, y )
call dsctr(nz, x, indx, y )
call csctr(nz, x, indx, y )
call zsctr(nz, x, indx, y )
call sctr(x, indx, y)
```

Include Files

- mkl.fi, blas.f90

Description

The ?sctr routines scatter the elements of the compressed sparse vector (*nz*, *x*, *indx*) to a full-storage vector *y*. The routines modify only the elements of *y* whose indices are listed in the array *indx*:

$y(\text{indx}(i)) = x(i)$, for $i=1,2,\dots,nz$.

Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>x</i> to be scattered.
<i>indx</i>	INTEGER. Specifies indices of elements to be scattered. Array, size at least <i>nz</i> .
<i>x</i>	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least <i>nz</i> . Contains the vector to be converted to full-storage form.

Output Parameters

<i>y</i>	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least $\max(\text{indx}(i))$. Contains the vector <i>y</i> with updated elements.
----------	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `sctr` interface are the following:

<code>x</code>	Holds the vector with the number of elements <code>nz</code> .
<code>indx</code>	Holds the vector with the number of elements <code>nz</code> .
<code>y</code>	Holds the vector with the number of elements <code>nz</code> .

Sparse BLAS Level 2 and Level 3 Routines

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® Math Kernel Library (Intel® MKL). Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel MKL Sparse BLAS Level 2 and Level 3 routines are discussed in the [Linear Solvers Basics](#) appendix.

The Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for [Iterative Sparse Solvers based on Reverse Communication Interface \(RCI ISS\)](#) described in the Chapter 8 of the manual.

Intel MKL provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [[Rem05](#)].

Some software packages and libraries (the [PARDISO* Solver](#) used in Intel MKL, *Sparskit 2* [[Saad94](#)], the Compaq* Extended Math Library (CXML)[[CXML01](#)]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel MKL provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section [One-based and Zero-based Indexing](#)).

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix `mkl_` or `mkl_cspblas_`.

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character> <data> <operation>( )
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character> <data> <mtype> <operation>( )
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data><mtype><operation>( )
```

for routines with zero-based indexing.

The `<character>` field indicates the data type:

<code>s</code>	real, single precision
----------------	------------------------

c	complex, single precision
d	real, double precision
z	complex, double precision

The `<data>` field indicates the sparse matrix storage format (see section [Sparse Matrix Storage Formats](#)):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The `<operation>` field indicates the type of operation:

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sv	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

The field `<mtype>` indicates the matrix type:

ge	sparse representation of a general matrix
sy	sparse representation of the upper or lower triangle of a symmetric matrix
tr	sparse representation of a triangular matrix

Sparse Matrix Storage Formats

The current version of Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following point entry [[Duff86](#)] storage formats for sparse matrices:

- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:

- *block sparse row* format (BSR) and its variations.

For more information see "[Sparse Matrix Storage Formats](#)" in Appendix A.

Intel MKL provides auxiliary routines - [matrix converters](#) - that convert sparse matrix from one storage format to another.

Routines and Supported Operations

This section describes operations supported by the Intel MKL Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:

A is a sparse matrix;
 B and C are dense matrices;
 D is a diagonal scaling matrix;
 x and y are dense vectors;
 α and β are scalars;

$\text{op}(A)$ is one of the possible operations:

$\text{op}(A) = A$;
 $\text{op}(A) = A'$ - transpose of A ;
 $\text{op}(A) = \text{conj}(A')$ - conjugated transpose of A .

$\text{inv}(\text{op}(A))$ denotes the inverse of $\text{op}(A)$.

The Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between sparse matrix and dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Intel MKL provides an additional set of the Sparse BLAS Level 2 routines with *simplified interfaces*. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):

```
y := op(A)*x
```

- solving a single triangular system (for triangular matrices):

```
y := inv(op(A))*x
```

Matrix type is indicated by the field `<mtype>` in the routine name (see section [Naming Conventions in Sparse BLAS Level 2 and Level 3](#)).

NOTE

The routines with simplified interfaces support only four sparse matrix storage formats, specifically:

- CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;
- diagonal format accepted in the CXML;
- coordinate format;
- BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

The Intel MKL Sparse BLAS Level 2 and Level 3 routines do not support in-place operations.

Complete list of all routines is given in the ["Sparse BLAS Level 2 and Level 3 Routines"](#).

Interface Consideration

One-Based and Zero-Based Indexing

The Intel MKL Sparse BLAS Level 2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of [Sparse BLAS Level 2 and Level 3 Routines](#).

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array x can be presented in case of one-based indexing as follows:

Element index: 1 2 3 4 5

Element value: 1.0 5.0 7.0 8.0 9.0

and in case of zero-based indexing as follows:

Element index: 0 1 2 3 4

Element value: 1.0 5.0 7.0 8.0 9.0

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the "[Sparse Matrix Storage Formats](#)" in Appendix A.

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.

Parameter	One-based Indexing	Zero-based Indexing
<i>val</i>	Array containing non-zero elements of the matrix A , its length is $pntrb(m) - pntrb(1)$.	Array containing non-zero elements of the matrix A , its length is $pntrb(m-1) - pntrb(0)$.
<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(1)+1$ is the first index of row i in the arrays <i>val</i> and <i>indx</i>	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(1)$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(0)-1$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one.	Array of length $m+1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m)$ is equal to the number of non-zeros.
<i>ldb</i>	Specifies the leading dimension of b as declared in the calling (sub)program.	Specifies the second dimension of b as declared in the calling (sub)program.

Parameter	One-based Indexing	Zero-based Indexing
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>c</i> as declared in the calling (sub)program.

Differences Between Intel MKL and NIST* Interfaces

The Intel MKL Sparse BLAS Level 3 routines have the following conventional interfaces:

`mkl_xyyyymm(transa, m, n, k, alpha, matdescra, arg(A), b, ldb, beta, c, ldc)`, for matrix-matrix product;

`mkl_xyyysm(transa, m, n, alpha, matdescra, arg(A), b, ldb, c, ldc)`, for triangular solvers with multiple right-hand sides.

Here *x* denotes data type, and *yyy* - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

`xyyyymm(transa, m, n, k, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for matrix-matrix product;

`xyyyysm(transa, m, n, unitd, dv, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument *transa* indicates what operation is performed and is slightly different in the NSB library (see Table "Parameter *transa*"). The arguments *m* and *k* are the number of rows and column in the matrix *A*, respectively, *n* is the number of columns in the matrix *C*. The arguments *alpha* and *beta* are scalar *alpha* and *beta* respectively (*beta* is not used in the Intel MKL triangular solvers.) The arguments *b* and *c* are rectangular arrays with the leading dimension *ldb* and *ldc*, respectively. `arg(A)` denotes the list of arguments that describe the sparse representation of *A*.

Parameter *transa*

	MKL interface	NSB interface	Operation
data type	CHARACTER*1	INTEGER	
value	N or n	0	$op(A) = A$
	T or t	1	$op(A) = A'$
	C or c	2	$op(A) = A'$

Parameter *matdescra*

The parameter *matdescra* describes the relevant characteristic of the matrix *A*. This manual describes *matdescra* as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel MKL Sparse BLAS routines. Elements *matdescra*(5) and *matdescra*(6) are reserved for future use. Note that whether *matdescra* is described in your application as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel MKL routines. To learn more about declaration of the *matdescra* array, see the Sparse BLAS examples located in the Intel MKL installation directory: `examples/spblasf/` for Fortran . The table below lists elements of the parameter *matdescra*, their Fortran values, and their meanings. The parameter *matdescra* corresponds to the argument *descra* from NSB library.

Possible Values of the Parameter *matdescra* (*descra*)

	MKL interface		NSB interface	Matrix characteristics
	one-based indexing	zero-based indexing		
data type	CHARACTER	Char	INTEGER	
1st element value	<i>matdescra</i> (1)	<i>matdescra</i> (0)	<i>descra</i> (1)	matrix structure
	G	G	0	general
	S	S	1	symmetric ($A = A'$)
	H	H	2	Hermitian ($A = \text{conjg}(A')$)
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric ($A = -A'$)
	D	D	5	diagonal
2nd element value	<i>matdescra</i> (2)	<i>matdescra</i> (1)	<i>descra</i> (2)	upper/lower triangular indicator
	L	L	1	lower
	U	U	2	upper
3rd element value	<i>matdescra</i> (3)	<i>matdescra</i> (2)	<i>descra</i> (3)	main diagonal type
	N	N	0	non-unit
	U	U	1	unit
4th element value	<i>matdescra</i> (4)	<i>matdescra</i> (3)	<i>descra</i> (4)	type of indexing
	F		1	one-based indexing
		C	0	zero-based indexing

In some cases possible element values of the parameter *matdescra* depend on the values of other elements. The [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#) lists all possible combinations of element values for both multiplication routines and triangular solvers.

Possible Combinations of Element Values of the Parameter *matdescra*

Routines	<i>matdescra</i> (1)	<i>matdescra</i> (2)	<i>matdescra</i> (3)	<i>matdescra</i> (4)
Multiplication Routines	G	ignored	ignored	F (default) or C
	S or H	L (default)	N (default)	F (default) or C
	S or H	L (default)	U	F (default) or C
	S or H	U	N (default)	F (default) or C
	S or H	U	U	F (default) or C
	A	L (default)	ignored	F (default) or C
	A	U	ignored	F (default) or C
Multiplication Routines and Triangular Solvers	T	L	U	F (default) or C

Routines	matdescra(1)	matdescra(2)	matdescra(3)	matdescra(4)
	T	L	N	F (default) or C
	T	U	U	F (default) or C
	T	U	N	F (default) or C
	D	ignored	N (default)	F (default) or C
	D	ignored	U	F (default) or C

For a matrix in the skyline format with the main diagonal declared to be a unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats, diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

Operations with Partial Matrices

One of the distinctive feature of the Intel MKL Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and the main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter *matdescra*.

An arbitrary sparse matrix A can be decomposed as

$$A = L + D + U$$

where L is the strict lower triangle of A , U is the strict upper triangle of A , D is the main diagonal.

Table "Output Matrices for Multiplication Routines" shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix A for multiplication routines.

Output Matrices for Multiplication Routines

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
G	ignored	ignored	$\alpha * \text{op}(A) * x + \beta * y$ $\alpha * \text{op}(A) * B + \beta * C$
S or H	L	N	$\alpha * \text{op}(L+D+L') * x + \beta * y$ $\alpha * \text{op}(L+D+L') * B + \beta * C$
S or H	L	U	$\alpha * \text{op}(L+I+L') * x + \beta * y$ $\alpha * \text{op}(L+I+L') * B + \beta * C$
S or H	U	N	$\alpha * \text{op}(U'+D+U) * x + \beta * y$ $\alpha * \text{op}(U'+D+U) * B + \beta * C$
S or H	U	U	$\alpha * \text{op}(U'+I+U) * x + \beta * y$ $\alpha * \text{op}(U'+I+U) * B + \beta * C$
T	L	U	$\alpha * \text{op}(L+I) * x + \beta * y$ $\alpha * \text{op}(L+I) * B + \beta * C$
T	L	N	$\alpha * \text{op}(L+D) * x + \beta * y$ $\alpha * \text{op}(L+D) * B + \beta * C$
T	U	U	$\alpha * \text{op}(U+I) * x + \beta * y$ $\alpha * \text{op}(U+I) * B + \beta * C$
T	U	N	$\alpha * \text{op}(U+D) * x + \beta * y$ $\alpha * \text{op}(U+D) * B + \beta * C$
A	L	ignored	$\alpha * \text{op}(L-L') * x + \beta * y$

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
			$\alpha * \text{op}(L-L') * B + \beta * C$
A	U	ignored	$\alpha * \text{op}(U-U') * x + \beta * y$ $\alpha * \text{op}(U-U') * B + \beta * C$
D	ignored	N	$\alpha * D * x + \beta * y$ $\alpha * D * B + \beta * C$
D	ignored	U	$\alpha * x + \beta * y$ $\alpha * B + \beta * C$

Table “Output Matrices for Triangular Solvers” shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix A for triangular solvers.

Output Matrices for Triangular Solvers

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	$\alpha * \text{inv}(\text{op}(L+L)) * x$ $\alpha * \text{inv}(\text{op}(L+L)) * B$
T	L	U	$\alpha * \text{inv}(\text{op}(L+L)) * x$ $\alpha * \text{inv}(\text{op}(L+L)) * B$
T	U	N	$\alpha * \text{inv}(\text{op}(U+U)) * x$ $\alpha * \text{inv}(\text{op}(U+U)) * B$
T	U	U	$\alpha * \text{inv}(\text{op}(U+U)) * x$ $\alpha * \text{inv}(\text{op}(U+U)) * B$
D	ignored	N	$\alpha * \text{inv}(D) * x$ $\alpha * \text{inv}(D) * B$
D	ignored	U	$\alpha * x$ $\alpha * B$

Sparse BLAS Level 2 and Level 3 Routines.

Table “Sparse BLAS Level 2 and Level 3 Routines” lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description
Simplified interface, one-based indexing	
<code>mkl_?csrgevmv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)
<code>mkl_?bsrgevmv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).
<code>mkl_?coogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format.

Routine/Function	Description
<code>mkl_?diagmv</code>	Computes matrix - vector product of a sparse general matrix in the diagonal format.
<code>mkl_?csrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)
<code>mkl_?bsrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).
<code>mkl_?coosvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.
<code>mkl_?diasvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.
<code>mkl_?csrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).
<code>mkl_?bsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).
<code>mkl_?cootrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
<code>mkl_?diatrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.

Simplified interface, zero-based indexing

<code>mkl_cspblas_?csrsvmv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrsvmv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?coosvmv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing
<code>mkl_cspblas_?bsrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?coosvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?cootrsv</code>	Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.

Routine/Function	Description
Typical (conventional) interface, one-based and zero-based indexing	
mkl_?csr _{mv}	Computes matrix - vector product of a sparse matrix in the CSR format.
mkl_?bsr _{mv}	Computes matrix - vector product of a sparse matrix in the BSR format.
mkl_?csc _{mv}	Computes matrix - vector product for a sparse matrix in the CSC format.
mkl_?coo _{mv}	Computes matrix - vector product for a sparse matrix in the coordinate format.
mkl_?csr _{sv}	Solves a system of linear equations for a sparse matrix in the CSR format.
mkl_?bsr _{sv}	Solves a system of linear equations for a sparse matrix in the BSR format.
mkl_?csc _{sv}	Solves a system of linear equations for a sparse matrix in the CSC format.
mkl_?coo _{sv}	Solves a system of linear equations for a sparse matrix in the coordinate format.
mkl_?csr _{mm}	Computes matrix - matrix product of a sparse matrix in the CSR format
mkl_?bsr _{mm}	Computes matrix - matrix product of a sparse matrix in the BSR format.
mkl_?csc _{mm}	Computes matrix - matrix product of a sparse matrix in the CSC format
mkl_?coo _{mm}	Computes matrix - matrix product of a sparse matrix in the coordinate format.
mkl_?csr _{sm}	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
mkl_?bsr _{sm}	Solves a system of linear matrix equations for a sparse matrix in the BSR format.
mkl_?csc _{sm}	Solves a system of linear matrix equations for a sparse matrix in the CSC format.
mkl_?coo _{sm}	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Typical (conventional) interface, one-based indexing

mkl_?diam _{mv}	Computes matrix - vector product of a sparse matrix in the diagonal format.
mkl_?skym _{mv}	Computes matrix - vector product for a sparse matrix in the skyline storage format.
mkl_?dias _{sv}	Solves a system of linear equations for a sparse matrix in the diagonal format.

Routine/Function	Description
<code>mkl_?skysv</code>	Solves a system of linear equations for a sparse matrix in the skyline format.
<code>mkl_?diamm</code>	Computes matrix - matrix product of a sparse matrix in the diagonal format.
<code>mkl_?skymm</code>	Computes matrix - matrix product of a sparse matrix in the skyline storage format.
<code>mkl_?diasm</code>	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
<code>mkl_?skysm</code>	Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.
Auxiliary routines	
Matrix converters	
<code>mkl_?dnscsr</code>	Converts a sparse matrix in uncompressed representation to CSR format (3-array variation) and vice versa.
<code>mkl_?csrcoo</code>	Converts a sparse matrix in CSR format (3-array variation) to coordinate format and vice versa.
<code>mkl_?csrbsr</code>	Converts a sparse matrix in CSR format to BSR format (3-array variations) and vice versa.
<code>mkl_?csrsc</code>	Converts a sparse matrix in CSR format to CSC format and vice versa (3-array variations).
<code>mkl_?csrdia</code>	Converts a sparse matrix in CSR format (3-array variation) to diagonal format and vice versa.
<code>mkl_?csrsky</code>	Converts a sparse matrix in CSR format (3-array variation) to sky line format and vice versa.
Operations on sparse matrices	
<code>mkl_?csradd</code>	Computes the sum of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
<code>mkl_?csmultcsr</code>	Computes the product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
<code>mkl_?csmultd</code>	Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

`mkl_?csrgemv`

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_scsrgemv(transa, m, a, ia, ja, x, y)
call mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
call mkl_ccsrgemv(transa, m, a, ia, ja, x, y)
call mkl_zcsrgemv(transa, m, a, ia, ja, x, y)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrsgemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation), A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := A'*x$,</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix A.</p>
<i>a</i>	<p>REAL for <code>mkl_scsrsgemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrsgemv</code>.</p> <p>COMPLEX for <code>mkl_ccsrsgemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrsgemv</code>.</p> <p>Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ia</i>	<p>INTEGER. Array of length $m + 1$, containing indices of elements in the array a, such that $ia(i)$ is the index in the array a of the first non-zero element from the row i. The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the length of the array a. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scsrsgemv</code>.</p>

DOUBLE PRECISION for `mkl_dcsrgemv`.

COMPLEX for `mkl_ccsrgemv`.

DOUBLE COMPLEX for `mkl_zcsrgemv`.

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y REAL for `mkl_scsrgemv`.

DOUBLE PRECISION for `mkl_dcsrgemv`.

COMPLEX for `mkl_ccsrgemv`.

DOUBLE COMPLEX for `mkl_zcsrgemv`.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrgemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation), A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$,
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL for <code>mkl_sbsrgemv</code> . DOUBLE PRECISION for <code>mkl_dbsrgemv</code> . COMPLEX for <code>mkl_cbsrgemv</code> . DOUBLE COMPLEX for <code>mkl_zbsrgemv</code> .

Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to *values* array description in [BSR Format](#) for more details.

ia INTEGER. Array of length $(m + 1)$, containing indices of block in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in [BSR Format](#) for more details.

ja INTEGER. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks of the matrix A . Refer to *columns* array description in [BSR Format](#) for more details.

x REAL for `mkl_sbsrgemv`.
DOUBLE PRECISION for `mkl_dbsrgemv`.
COMPLEX for `mkl_cbsrgemv`.
DOUBLE COMPLEX for `mkl_zbsrgemv`.

Array, size $(m*lb)$.

On entry, the array x must contain the vector x .

Output Parameters

y REAL for `mkl_sbsrgemv`.
DOUBLE PRECISION for `mkl_dbsrgemv`.
COMPLEX for `mkl_cbsrgemv`.
DOUBLE COMPLEX for `mkl_zbsrgemv`.

Array, size at least $(m*lb)$.

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_?coogemv

Computes matrix-vector product of a sparse general matrix stored in the coordinate format with one-based indexing.

Syntax

```
call mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?coogemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$,</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	<p>REAL for <code>mkl_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoogemv</code>.</p> <p>COMPLEX for <code>mkl_ccoogemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoogemv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoogemv</code>.</p> <p>COMPLEX for <code>mkl_ccoogemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoogemv</code>.</p> <p>Array, size is <i>m</i>.</p> <p>One entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoogemv</code>.</p> <p>COMPLEX for <code>mkl_ccoogemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoogemv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>On exit, the array <i>y</i> must contain the vector <i>y</i>.</p>
----------	---

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?diagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_ddiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_cdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_zdiagemv(transa, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?diagemv` routine performs a matrix-vector operation defined as

`y := A*x`

or

`y := A'*x,`

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the diagonal storage format, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the operation. If <code>transa = 'N' or 'n'</code> , then <code>y := A*x</code> If <code>transa = 'T' or 't' or 'C' or 'c'</code> , then <code>y := A'*x,</code>
<code>m</code>	INTEGER. Number of rows of the matrix A .
<code>val</code>	REAL for <code>mkl_sdiagemv</code> . DOUBLE PRECISION for <code>mkl_ddiagemv</code> . COMPLEX for <code>mkl_ccsrgemv</code> . DOUBLE COMPLEX for <code>mkl_zdiagemv</code> . Two-dimensional array of size <code>lval*ndiag</code> , contains non-zero diagonals of the matrix A . Refer to <code>values</code> array description in Diagonal Storage Scheme for more details.
<code>lval</code>	INTEGER. Leading dimension of <code>val</code> $lval \geq m$. Refer to <code>lval</code> description in Diagonal Storage Scheme for more details.
<code>idiag</code>	INTEGER. Array of length <code>ndiag</code> , contains the distances between main diagonal and each non-zero diagonals in the matrix A . Refer to <code>distance</code> array description in Diagonal Storage Scheme for more details.
<code>ndiag</code>	INTEGER. Specifies the number of non-zero diagonals of the matrix A .
<code>x</code>	REAL for <code>mkl_sdiagemv</code> . DOUBLE PRECISION for <code>mkl_ddiagemv</code> . COMPLEX for <code>mkl_ccsrgemv</code> . DOUBLE COMPLEX for <code>mkl_zdiagemv</code> . Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y

REAL for mkl_sdiagmv.
 DOUBLE PRECISION for mkl_ddiagmv.
 COMPLEX for mkl_ccsrgmv.
 DOUBLE COMPLEX for mkl_zdiagmv.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiagmv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiagmv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiagmv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiagmv(transa, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

mkl_?csrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_scsrsvmv(uplo, m, a, ia, ja, x, y)
call mkl_dcsrsvmv(uplo, m, a, ia, ja, x, y)
call mkl_ccsrsvmv(uplo, m, a, ia, ja, x, y)
call mkl_zcsrsvmv(uplo, m, a, ia, ja, x, y)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrsvmv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	<p>REAL for <code>mkl_scsrsvmv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrsvmv</code>.</p> <p>COMPLEX for <code>mkl_ccsrsvmv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrsvmv</code>.</p> <p>Array containing non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ia</i>	<p>INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i>, such that <i>ia</i>(<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

x REAL for `mkl_scsrsvmv`.
 DOUBLE PRECISION for `mkl_dcscrsvmv`.
 COMPLEX for `mkl_ccscrsvmv`.
 DOUBLE COMPLEX for `mkl_zcscrsvmv`.

Array, size is *m*.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for `mkl_scsrsvmv`.
 DOUBLE PRECISION for `mkl_dcscrsvmv`.
 COMPLEX for `mkl_ccscrsvmv`.
 DOUBLE COMPLEX for `mkl_zcscrsvmv`.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcscrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccscrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
  CHARACTER*1  uplo
```

```
  INTEGER      m
```

```
  INTEGER      ia(*), ja(*)
```

```
  DOUBLE COMPLEX  a(*), x(*), y(*)
```

mkl_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

uplo CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix A is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix A is used.

m INTEGER. Number of block rows of the matrix A .

<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL for <code>mkl_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_dbsrsymv</code> . COMPLEX for <code>mkl_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_zcsrgemv</code> . Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL for <code>mkl_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_dbsrsymv</code> . COMPLEX for <code>mkl_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_zcsrgemv</code> . Array, size $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL for <code>mkl_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_dbsrsymv</code> . COMPLEX for <code>mkl_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_zcsrgemv</code> . Array, size at least $(m*lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  COMPLEX     a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing.

Syntax

```
call mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?coosymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix <i>A</i>.</p>
<i>val</i>	<p>REAL for <code>mkl_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoosymv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoosymv</code>.</p> <p>Array, size is <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoosymv</code>.</p> <p>Array, size at least <i>m</i>.</p>
----------	--

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cdcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?diasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_cdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
call mkl_zdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?diasymv` routine performs a matrix-vector operation defined as

$$y := A*x$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.</p>
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>val</i>	<p>REAL for <code>mkl_sdiasymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddiasymv</code>.</p> <p>COMPLEX for <code>mkl_cdiasymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdiasymv</code>.</p> <p>Two-dimensional array of size <i>lval</i> by <i>ndiag</i>, contains non-zero diagonals of the matrix A. Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.</p>
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	<p>INTEGER. Array of length <i>ndiag</i>, contains the distances between main diagonal and each non-zero diagonals in the matrix A.</p> <p>Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.</p>
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix A .
<i>x</i>	<p>REAL for <code>mkl_sdiasymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddiasymv</code>.</p> <p>COMPLEX for <code>mkl_cdiasymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdiasymv</code>.</p> <p>Array, size is <i>m</i>.</p>

On entry, the array x must contain the vector x .

Output Parameters

y REAL for `mkl_sdiasymv`.
 DOUBLE PRECISION for `mkl_ddiasymv`.
 COMPLEX for `mkl_cdiasymv`.
 DOUBLE COMPLEX for `mkl_zdiasymv`.
 Array, size at least m .
 On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiasymv(uplo, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

`mkl_?csrsv`

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is a unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>a</i>	REAL for <code>mkl_scsrtrmv</code> . DOUBLE PRECISION for <code>mkl_dcsrtrmv</code> .

COMPLEX for `mkl_ccsrtrmv`.

DOUBLE COMPLEX for `mkl_zcsrtrmv`.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia INTEGER. Array of length $m + 1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array a . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

x REAL for `mkl_scsrtrmv`.
 DOUBLE PRECISION for `mkl_dcsrtrmv`.
 COMPLEX for `mkl_ccsrtrmv`.
 DOUBLE COMPLEX for `mkl_zcsrtrmv`.

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y REAL for `mkl_scsrtrmv`.
 DOUBLE PRECISION for `mkl_dcsrtrmv`.
 COMPLEX for `mkl_ccsrtrmv`.
 DOUBLE COMPLEX for `mkl_zcsrtrmv`.

Array, size at least m .

Contains the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) :

$$y := A*x$$

or

$$y := A'*x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$.
<i>diag</i>	CHARACTER*1. Specifies whether A is a unit triangular matrix. If <i>diag</i> = 'U' or 'u', then A is a unit triangular. If <i>diag</i> = 'N' or 'n', then A is not a unit triangular.
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL for <code>mkl_sbsrtrsv</code> . DOUBLE PRECISION for <code>mkl_dbsrtrsv</code> . COMPLEX for <code>mkl_cbsrtrsv</code> . DOUBLE COMPLEX for <code>mkl_zbsrtrsv</code> . Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	<p>INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i>, such that $ia(I)$ is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i>. The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.</p>
<i>ja</i>	<p>INTEGER.</p> <p>Array containing the column indices for each non-zero block in the matrix <i>A</i>. Its length is equal to the number of non-zero blocks of the matrix <i>A</i>. Refer to <i>columns</i> array description in BSR Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_sbsrtrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dbsrtrsv</code>.</p> <p>COMPLEX for <code>mkl_cbsrtrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zbsrtrsv</code>.</p> <p>Array, size $(m * lb)$.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_sbsrtrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dbsrtrsv</code>.</p> <p>COMPLEX for <code>mkl_cbsrtrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zbsrtrsv</code>.</p> <p>Array, size at least $(m * lb)$.</p> <p>On exit, the array <i>y</i> must contain the vector <i>y</i>.</p>
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  COMPLEX     a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER     m, lb
```

```
  INTEGER     ia(*), ja(*)
```

```
  DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing.

Syntax

```
call mkl_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?cootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

```
A*y = x
```

or

```
A'*y = x,
```

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is considered.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.</p>
<i>transa</i>	<p>CHARACTER*1. Specifies the system of linear equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $A*y = x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,</p>
<i>diag</i>	<p>CHARACTER*1. Specifies whether <i>A</i> is unit triangular.</p> <p>If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular.</p> <p>If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix <i>A</i>.</p>
<i>val</i>	<p>REAL for <code>mkl_scootrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcootrsv</code>.</p> <p>COMPLEX for <code>mkl_ccootrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcootrsv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scootrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcootrsv</code>.</p> <p>COMPLEX for <code>mkl_ccootrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcootrsv</code>.</p>

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y REAL for `mkl_scootrsv`.
 DOUBLE PRECISION for `mkl_dcootrsv`.
 COMPLEX for `mkl_ccootrsv`.
 DOUBLE COMPLEX for `mkl_zcootrsv`.
 Array, size at least m .
 Contains the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?diatrsv

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_cdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
call mkl_zdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?diatrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <code>uplo = 'U' or 'u'</code> , then the upper triangle of the matrix A is used. If <code>uplo = 'L' or 'l'</code> , then the low triangle of the matrix A is used.
<code>transa</code>	CHARACTER*1. Specifies the system of linear equations. If <code>transa = 'N' or 'n'</code> , then $A*y = x$ If <code>transa = 'T' or 't' or 'C' or 'c'</code> , then $A'*y = x$,
<code>diag</code>	CHARACTER*1. Specifies whether A is unit triangular. If <code>diag = 'U' or 'u'</code> , then A is unit triangular. If <code>diag = 'N' or 'n'</code> , then A is not unit triangular.

<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	REAL for <code>mkl_sdiatrsv</code> . DOUBLE PRECISION for <code>mkl_ddiatrsv</code> . COMPLEX for <code>mkl_cdiatrsv</code> . DOUBLE COMPLEX for <code>mkl_zdiatrsv</code> . Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL for <code>mkl_sdiatrsv</code> . DOUBLE PRECISION for <code>mkl_ddiatrsv</code> . COMPLEX for <code>mkl_cdiatrsv</code> . DOUBLE COMPLEX for <code>mkl_zdiatrsv</code> . Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL for <code>mkl_sdiatrsv</code> . DOUBLE PRECISION for <code>mkl_ddiatrsv</code> . COMPLEX for <code>mkl_cdiatrsv</code> . DOUBLE COMPLEX for <code>mkl_zdiatrsv</code> . Array, size at least <i>m</i> . Contains the vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiatrsv(uplo, transa, diag, m, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

mkl_cspblas_?csrgev

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_scsrgev(transa, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_dcsrgev(transa, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_ccsrgev(transa, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_zcsrgev(transa, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?csrsgmv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation) with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$,</p>
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>a</i>	<p>REAL for <code>mkl_cspblas_scsrsgmv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcsrsgmv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccsrsgmv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcsrsgmv</code>.</p> <p>Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ia</i>	<p>INTEGER. Array of length $m + 1$, containing indices of elements in the array a, such that $ia(I)$ is the index in the array a of the first non-zero element from the row I. The value of the last element $ia(m)$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the length of the array a. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_cspblas_scsrsgmv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcsrsgmv</code>.</p>

COMPLEX for mkl_cspblas_ccsrgemv.

DOUBLE COMPLEX for mkl_cspblas_zcsrgemv.

Array, size is m .

One entry, the array x must contain the vector x .

Output Parameters

y

REAL for mkl_cspblas_scsrgemv.

DOUBLE PRECISION for mkl_cspblas_dcsrgemv.

COMPLEX for mkl_cspblas_ccsrgemv.

DOUBLE COMPLEX for mkl_cspblas_zcsrgemv.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcsrgemv(transa, m, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_cspblas_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?bsrgemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$,
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>a</i>	REAL for <code>mkl_cspblas_sbsrgemv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrgemv</code> . COMPLEX for <code>mkl_cspblas_cbsrgemv</code> .

DOUBLE COMPLEX for `mkl_cspblas_zbsrgemv`.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *lb*lb*. Refer to *values* array description in [BSR Format](#) for more details.

ia INTEGER. Array of length $(m + 1)$, containing indices of block in the array *a*, such that *ia*(*i*) is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element *ia*($m + 1$) is equal to the number of non-zero blocks. Refer to *rowIndex* array description in [BSR Format](#) for more details.

ja INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in [BSR Format](#) for more details.

x REAL for `mkl_cspblas_sbsrgemv`.
 DOUBLE PRECISION for `mkl_cspblas_dbsrgemv`.
 COMPLEX for `mkl_cspblas_cbsrgemv`.
 DOUBLE COMPLEX for `mkl_cspblas_zbsrgemv`.
 Array, size $(m*lb)$.
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for `mkl_cspblas_sbsrgemv`.
 DOUBLE PRECISION for `mkl_cspblas_dbsrgemv`.
 COMPLEX for `mkl_cspblas_cbsrgemv`.
 DOUBLE COMPLEX for `mkl_cspblas_zbsrgemv`.
 Array, size at least $(m*lb)$.
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 transa
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1  transa
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  DOUBLE PRECISION  a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1  transa
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  COMPLEX      a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1  transa
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  DOUBLE COMPLEX  a(*), x(*), y(*)
```

mkl_cspblas_?coogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing.

Syntax

```
call mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_dcoogemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$.</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix <i>A</i>.</p>
<i>val</i>	<p>REAL for <code>mkl_cspblas_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoogemv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccoogemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcoogemv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_cspblas_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoogemv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccoogemv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcoogemv</code>.</p> <p>Array, size is <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_cspblas_scoogemv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoogemv</code>.</p>
----------	---

COMPLEX for `mkl_cspblas_ccoogemv`.

DOUBLE COMPLEX for `mkl_cspblas_zcoogemv`.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1  transa
```

```
INTEGER      m, nnz
```

```
INTEGER      rowind(*), colind(*)
```

```
REAL         val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1  transa
```

```
INTEGER      m, nnz
```

```
INTEGER      rowind(*), colind(*)
```

```
DOUBLE PRECISION  val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1  transa
```

```
INTEGER      m, nnz
```

```
INTEGER      rowind(*), colind(*)
```

```
COMPLEX      val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1  transa
```

```
INTEGER      m, nnz
```

```
INTEGER      rowind(*), colind(*)
```

```
DOUBLE COMPLEX  val(*), x(*), y(*)
```

`mkl_cspblas_?csrsvmv`

Computes matrix-vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_scsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_dcsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_ccsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_zcsrsvmv(uplo, m, a, ia, ja, x, y)
```

Include Files

- `mkl.fi`

Description

The `mkl_cspblas_?csrsvmv` routine performs a matrix-vector operation defined as

$$y := A*x$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation) with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix A.</p>
<i>a</i>	<p>REAL for <code>mkl_cspblas_scsrsvmv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcsrsvmv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccsrsvmv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcsrsvmv</code>.</p> <p>Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ia</i>	<p>INTEGER. Array of length $m + 1$, containing indices of elements in the array a, such that $ia(i)$ is the index in the array a of the first non-zero element from the row i. The value of the last element $ia(m + 1)$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the length of the array a. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>

x REAL for mkl_cspblas_scsrsvmv.
 DOUBLE PRECISION for mkl_cspblas_dcsrsvmv.
 COMPLEX for mkl_cspblas_ccsrsvmv.
 DOUBLE COMPLEX for mkl_cspblas_zcsrsvmv.
 Array, size is *m*.
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_cspblas_scsrsvmv.
 DOUBLE PRECISION for mkl_cspblas_dcsrsvmv.
 COMPLEX for mkl_cspblas_ccsrsvmv.
 DOUBLE COMPLEX for mkl_cspblas_zcsrsvmv.
 Array, size at least *m*.
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcsrsvmv(uplo, m, a, ia, ja, x, y)
```

```
CHARACTER*1    uplo
```

```
INTEGER        m
```

```
INTEGER        ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_cspblas_?bsrsvmv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-arrays variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_sbsrsvmv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_dbsrsvmv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_cbsrsvmv(uplo, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_zbsrsvmv(uplo, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?bsrsvmv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

uplo CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix A is used.

If *uplo* = 'L' or 'l', then the low triangle of the matrix A is used.

m INTEGER. Number of block rows of the matrix A .

<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL for <code>mkl_cspblas_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrsymv</code> . COMPLEX for <code>mkl_cspblas_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zbsrsymv</code> . Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL for <code>mkl_cspblas_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrsymv</code> . COMPLEX for <code>mkl_cspblas_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zbsrsymv</code> . Array, size $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL for <code>mkl_cspblas_sbsrsymv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrsymv</code> . COMPLEX for <code>mkl_cspblas_cbsrsymv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zbsrsymv</code> . Array, size at least $(m*lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_cspblas_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing .

Syntax

```
call mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?coosymv` routine performs a matrix-vector operation defined as

```
 $y := A \cdot x$ 
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.</p>
<i>m</i>	<p>INTEGER. Number of rows of the matrix <i>A</i>.</p>
<i>val</i>	<p>REAL for <code>mkl_cspblas_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcoosymv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_cspblas_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcoosymv</code>.</p> <p>Array, size is <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_cspblas_scoosymv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcoosymv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccoosymv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcoosymv</code>.</p> <p>Array, size at least <i>m</i>.</p>
----------	--

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 uplo
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_cspblas_?csrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
call mkl_cspblas_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?csrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

$$A*y = x$$

or

$$A'*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether matrix A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>a</i>	REAL for <code>mkl_cspblas_scsrtrsv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dcsrtrsv</code> . COMPLEX for <code>mkl_cspblas_ccsrtrsv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zcsrtrsv</code> . Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia INTEGER. Array of length $m+1$, containing indices of elements in the array *a*, such that $ia(i)$ is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element $ia(m)$ is equal to the number of non-zeros. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

x REAL for `mkl_cspblas_scsrtrsv`.
 DOUBLE PRECISION for `mkl_cspblas_dcsrtrsv`.
 COMPLEX for `mkl_cspblas_ccsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zcsrtrsv`.
 Array, size is *m*.
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for `mkl_cspblas_scsrtrsv`.
 DOUBLE PRECISION for `mkl_cspblas_dcsrtrsv`.
 COMPLEX for `mkl_cspblas_ccsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zcsrtrsv`.
 Array, size at least *m*.
 Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_cspblas_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax

```
call mkl_cspblas_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
call mkl_cspblas_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?bsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	<p>CHARACTER*1. Specifies the upper or low triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.</p>
<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A'*x$.</p>
<i>diag</i>	<p>CHARACTER*1. Specifies whether matrix <i>A</i> is unit triangular or not.</p> <p>If <i>diag</i> = 'U' or 'u', <i>A</i> is unit triangular.</p> <p>If <i>diag</i> = 'N' or 'n', <i>A</i> is not unit triangular.</p>
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	<p>REAL for <code>mkl_cspblas_sbsrtrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dbsrtrsv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_cbsrtrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zbsrtrsv</code>.</p> <p>Array containing elements of non-zero blocks of the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.</p>

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	<p>INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i>, such that <i>ia</i>(<i>I</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i>. The value of the last element <i>ia</i>($m + 1$) is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.</p>
-----------	--

ja INTEGER. Array containing the column indices for each non-zero block in the matrix *A*.
 Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in [BSR Format](#) for more details.

x REAL for `mkl_cspblas_sbsrtrsv`.
 DOUBLE PRECISION for `mkl_cspblas_dbsrtrsv`.
 COMPLEX for `mkl_cspblas_cbsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zbsrtrsv`.
 Array, size $(m \cdot lb)$.
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for `mkl_cspblas_sbsrtrsv`.
 DOUBLE PRECISION for `mkl_cspblas_dbsrtrsv`.
 COMPLEX for `mkl_cspblas_cbsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zbsrtrsv`.
 Array, size at least $(m \cdot lb)$.
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  REAL         a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER      m, lb
```

```
  INTEGER      ia(*), ja(*)
```

```
  COMPLEX      a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
CHARACTER*1 uplo, transa, diag
```

```
INTEGER m, lb
```

```
INTEGER ia(*), ja(*)
```

```
DOUBLE COMPLEX a(*), x(*), y(*)
```

mkl_cspblas_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing .

Syntax

```
call mkl_cspblas_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

```
call mkl_cspblas_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_cspblas_?cootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

```
 $\bar{A}y = x$ 
```

or

```
 $\bar{A}'y = x,$ 
```

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

`uplo` CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered.

If `uplo = 'U'` or `'u'`, then the upper triangle of the matrix A is used.

If `uplo = 'L'` or `'l'`, then the low triangle of the matrix A is used.

<i>transa</i>	<p>CHARACTER*1. Specifies the system of linear equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $A*y = x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'*y = x$,</p>
<i>diag</i>	<p>CHARACTER*1. Specifies whether <i>A</i> is unit triangular.</p> <p>If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular.</p> <p>If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	<p>REAL for <code>mkl_cspblas_scootrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcootrsv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccootrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcootrsv</code>.</p> <p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_cspblas_scootrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcootrsv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccootrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcootrsv</code>.</p> <p>Array, size is <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>REAL for <code>mkl_cspblas_scootrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_cspblas_dcootrsv</code>.</p> <p>COMPLEX for <code>mkl_cspblas_ccootrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_cspblas_zcootrsv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>Contains the vector <i>y</i>.</p>
----------	---

Interfaces

FORTRAN 77:

SUBROUTINE mkl_cspblas_scootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)

CHARACTER*1 uplo, transa, diag

INTEGER m, nnz

INTEGER rowind(*), colind(*)

REAL val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)

CHARACTER*1 uplo, transa, diag

INTEGER m, nnz

INTEGER rowind(*), colind(*)

DOUBLE PRECISION val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)

CHARACTER*1 uplo, transa, diag

INTEGER m, nnz

INTEGER rowind(*), colind(*)

COMPLEX val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)

CHARACTER*1 uplo, transa, diag

INTEGER m, nnz

INTEGER rowind(*), colind(*)

DOUBLE COMPLEX val(*), x(*), y(*)

mkl_?csr_{mv}

Computes matrix - vector product of a sparse matrix stored in the CSR format.

Syntax

call mkl_scsr_{mv}(transa, m, k, alpha, matdescra, val, indx, pnt_{rb}, pnt_{re}, x, beta, y)

call mkl_dcsr_{mv}(transa, m, k, alpha, matdescra, val, indx, pnt_{rb}, pnt_{re}, x, beta, y)

call mkl_ccsr_{mv}(transa, m, k, alpha, matdescra, val, indx, pnt_{rb}, pnt_{re}, x, beta, y)

call mkl_zcsr_{mv}(transa, m, k, alpha, matdescra, val, indx, pnt_{rb}, pnt_{re}, x, beta, y)

Include Files

- mkl.fi

Description

The mkl_?csr_{mv} routine performs a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y$$

or

$$y := \alpha A'x + \beta y,$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- k sparse matrix in the CSR format, A' is the transpose of A .

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scsrnmv</code> . DOUBLE PRECISION for <code>mkl_dcsrnmv</code> . COMPLEX for <code>mkl_ccsrnmv</code> . DOUBLE COMPLEX for <code>mkl_zcsrnmv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scsrnmv</code> . DOUBLE PRECISION for <code>mkl_dcsrnmv</code> . COMPLEX for <code>mkl_ccsrnmv</code> . DOUBLE COMPLEX for <code>mkl_zcsrnmv</code> . Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is $pntrb(m) - pntrb(1)$. For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$. Refer to <i>values</i> array description in CSR Format for more details.

<i>indx</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>columns</i> array description in CSR Format for more details.</p>
<i>pntrb</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSR Format for more details.</p>
<i>pntrr</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that $pntrr(i) - pntrr(1)$ is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that $pntrr(i) - pntrr(0) - 1$ is the last index of row <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scsrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscrsv</code>.</p> <p>COMPLEX for <code>mkl_ccscrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscrsv</code>.</p> <p>Array, size at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>
<i>beta</i>	<p>REAL for <code>mkl_scsrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscrsv</code>.</p> <p>COMPLEX for <code>mkl_ccscrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscrsv</code>.</p> <p>Specifies the scalar <i>beta</i>.</p>
<i>y</i>	<p>REAL for <code>mkl_scsrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscrsv</code>.</p> <p>COMPLEX for <code>mkl_ccscrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscrsv</code>.</p> <p>Array, size at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i>.</p>

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmv(transa, m, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha, beta
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrmv(transa, m, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrmv(transa, m, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?bsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format.

Syntax

```
call mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

```
call mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

```
call mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

```
call mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrmv` routine performs a matrix-vector operation defined as

```
 $y := \alpha A x + \beta y$ 
```

or

```
 $y := \alpha A' x + \beta y,$ 
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* block sparse matrix in the BSR format, *A'* is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := \alpha A' x + \beta y,$
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of block columns of the matrix <i>A</i> .

<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_sbsrmv</code> . DOUBLE PRECISION for <code>mkl_dbsrmv</code> . COMPLEX for <code>mkl_cbsrmv</code> . DOUBLE COMPLEX for <code>mkl_zbsrmv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_sbsrmv</code> . DOUBLE PRECISION for <code>mkl_dbsrmv</code> . COMPLEX for <code>mkl_cbsrmv</code> . DOUBLE COMPLEX for <code>mkl_zbsrmv</code> . Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i> . For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i> . Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i> . For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i> . Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>x</i>	REAL for <code>mkl_sbsrmv</code> . DOUBLE PRECISION for <code>mkl_dbsrmv</code> .

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Array, size at least $(k*lb)$ if *transa* = 'N' or 'n', and at least $(m*lb)$ otherwise. On entry, the array *x* must contain the vector *x*.

beta REAL for mkl_sbsrmv.

DOUBLE PRECISION for mkl_dbsrmv.

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Specifies the scalar *beta*.

y REAL for mkl_sbsrmv.

DOUBLE PRECISION for mkl_dbsrmv.

COMPLEX for mkl_cbsrmv.

DOUBLE COMPLEX for mkl_zbsrmv.

Array, size at least $(m*lb)$ if *transa* = 'N' or 'n', and at least $(k*lb)$ otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha, beta
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
```

```
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```

SUBROUTINE mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, k, lb
INTEGER      indx(*), pntrb(m), pntre(m)
COMPLEX      alpha, beta
COMPLEX      val(*), x(*), y(*)

```

```

SUBROUTINE mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, k, lb
INTEGER      indx(*), pntrb(m), pntre(m)
DOUBLE COMPLEX  alpha, beta
DOUBLE COMPLEX  val(*), x(*), y(*)

```

mkl_?cscmv

Computes matrix-vector product for a sparse matrix in the CSC format.

Syntax

```

call mkl_scscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)

```

Include Files

- mkl.fi

Description

The `mkl_?cscmv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A' x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A'* is the transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A' * x + \beta * y$,</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for mkl_scscmv.</p> <p>DOUBLE PRECISION for mkl_dcscmv.</p> <p>COMPLEX for mkl_ccscmv.</p> <p>DOUBLE COMPLEX for mkl_zcscmv.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_scscmv.</p> <p>DOUBLE PRECISION for mkl_dcscmv.</p> <p>COMPLEX for mkl_ccscmv.</p> <p>DOUBLE COMPLEX for mkl_zcscmv.</p> <p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>For one-based indexing its length is $pntrb(k) - pntrb(1)$.</p> <p>For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.</p> <p>Refer to <i>values</i> array description in CSC Format for more details.</p>
<i>indx</i>	<p>INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>rows</i> array description in CSC Format for more details.</p>
<i>pntrb</i>	INTEGER. Array of length <i>k</i> .

For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column i in the arrays val and $indx$.

For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column i in the arrays val and $indx$.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntrE

INTEGER. Array of length k .

For one-based indexing this array contains column indices, such that $pntrE(i) - pntrb(1)$ is the last index of column i in the arrays val and $indx$.

For zero-based indexing this array contains column indices, such that $pntrE(i) - pntrb(1) - 1$ is the last index of column i in the arrays val and $indx$.

Refer to *pointerE* array description in [CSC Format](#) for more details.

x

REAL for `mkl_scscmv`.

DOUBLE PRECISION for `mkl_dcscmv`.

COMPLEX for `mkl_ccscmv`.

DOUBLE COMPLEX for `mkl_zcscmv`.

Array, size at least k if $transa = 'N'$ or $'n'$ and at least m otherwise. On entry, the array x must contain the vector x .

β

REAL for `mkl_scscmv`.

DOUBLE PRECISION for `mkl_dcscmv`.

COMPLEX for `mkl_ccscmv`.

DOUBLE COMPLEX for `mkl_zcscmv`.

Specifies the scalar β .

y

REAL for `mkl_scscmv`.

DOUBLE PRECISION for `mkl_dcscmv`.

COMPLEX for `mkl_ccscmv`.

DOUBLE COMPLEX for `mkl_zcscmv`.

Array, size at least m if $transa = 'N'$ or $'n'$ and at least k otherwise. On entry, the array y must contain the vector y .

Output Parameters

y

Overwritten by the updated vector y .

Interfaces

FORTRAN 77:

SUBROUTINE mkl_scscmv(transa, m, k, alpha, matdescra, val, indx,

pntrb, pntre, x, beta, y)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

REAL alpha, beta

REAL val(*), x(*), y(*)

SUBROUTINE mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx,

pntrb, pntre, x, beta, y)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE PRECISION alpha, beta

DOUBLE PRECISION val(*), x(*), y(*)

SUBROUTINE mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx,

pntrb, pntre, x, beta, y)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

COMPLEX alpha, beta

COMPLEX val(*), x(*), y(*)

SUBROUTINE mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx,

pntrb, pntre, x, beta, y)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha, beta

DOUBLE COMPLEX val(*), x(*), y(*)

mkl_?coomv

Computes matrix - vector product for a sparse matrix in the coordinate format.

Syntax

```
call mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
call mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
call mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
call mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

Include Files

- mkl.fi

Description

The mkl_?coomv routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scoomv. DOUBLE PRECISION for mkl_dcoomv. COMPLEX for mkl_ccoomv. DOUBLE COMPLEX for mkl_zcoomv.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val

REAL for `mkl_scoomv`.
 DOUBLE PRECISION for `mkl_dcoomv`.
 COMPLEX for `mkl_ccoomv`.
 DOUBLE COMPLEX for `mkl_zcoomv`.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in [Coordinate Format](#) for more details.

rowind

INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*.

Refer to *columns* array description in [Coordinate Format](#) for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in [Coordinate Format](#) for more details.

x

REAL for `mkl_scoomv`.
 DOUBLE PRECISION for `mkl_dcoomv`.
 COMPLEX for `mkl_ccoomv`.
 DOUBLE COMPLEX for `mkl_zcoomv`.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for `mkl_scoomv`.
 DOUBLE PRECISION for `mkl_dcoomv`.
 COMPLEX for `mkl_ccoomv`.
 DOUBLE COMPLEX for `mkl_zcoomv`.

Specifies the scalar *beta*.

y

REAL for `mkl_scoomv`.
 DOUBLE PRECISION for `mkl_dcoomv`.
 COMPLEX for `mkl_ccoomv`.
 DOUBLE COMPLEX for `mkl_zcoomv`.

Array, size at least m if $transa = 'N'$ or $'n'$ and at least k otherwise. On entry, the array y must contain the vector y .

Output Parameters

y Overwritten by the updated vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL alpha, beta
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?csrsv

Solves a system of linear equations for a sparse matrix in the CSR format.

Syntax

```
call mkl_scsrsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_ccsrsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_zcsrsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?csrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A') \cdot x$,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scsrsv</code> . DOUBLE PRECISION for <code>mkl_dcsrsv</code> . COMPLEX for <code>mkl_ccsrsv</code> . DOUBLE COMPLEX for <code>mkl_zcsrsv</code> . Specifies the scalar <i>alpha</i> .

matdescra CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val REAL for `mkl_scsrsv`.
 DOUBLE PRECISION for `mkl_dcsrsv`.
 COMPLEX for `mkl_ccsrsv`.
 DOUBLE COMPLEX for `mkl_zcsrsv`.
 Array containing non-zero elements of the matrix *A*.
 For one-based indexing its length is $pntrb(m) - pntrb(1)$.
 For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.
 Refer to *values* array description in [CSR Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in [CSR Format](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSR Format](#) for more details.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1)$ is the last index of row *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0) - 1$ is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSR Format](#) for more details.

<i>x</i>	<p>REAL for <code>mkl_scsrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrsv</code>.</p> <p>COMPLEX for <code>mkl_ccsrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrsv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.</p>
<i>y</i>	<p>REAL for <code>mkl_scsrsv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrsv</code>.</p> <p>COMPLEX for <code>mkl_ccsrsv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrsv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>y</i> must contain the vector <i>y</i>. The elements are accessed with unit increment.</p>

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .
----------	-------------------------------------

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha
```

```
REAL val(*)
```

```
REAL x(*), y(*)
```

```
SUBROUTINE mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*)
```

```
DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_ccsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha
```

```
COMPLEX val(*)
```

```
COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*)
```

```
DOUBLE COMPLEX x(*), y(*)
```

mkl_?bsrsv

Solves a system of linear equations for a sparse matrix in the BSR format.

Syntax

```
call mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
call mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
call mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
call mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $y := \alpha * \text{inv}(A) * x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,</p>
<i>m</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for mkl_sbsrsv.</p> <p>DOUBLE PRECISION for mkl_dbsrsv.</p> <p>COMPLEX for mkl_cbsrsv.</p> <p>DOUBLE COMPLEX for mkl_zbsrsv.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_sbsrsv.</p> <p>DOUBLE PRECISION for mkl_dbsrsv.</p> <p>COMPLEX for mkl_cbsrsv.</p> <p>DOUBLE COMPLEX for mkl_zbsrsv.</p> <p>Array containing elements of non-zero blocks of the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.</p>

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	<p>INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i>.</p> <p>Its length is equal to the number of non-zero blocks in the matrix <i>A</i>.</p>
-------------	---

Refer to the *columns* array description in [BSR Format](#) for more details.

pntrb

INTEGER. Array of length *m*.

For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row *i* in the array *indx*.

For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row *i* in the array *indx*.

Refer to *pointerB* array description in [BSR Format](#) for more details.

pntrr

INTEGER. Array of length *m*.

For one-based indexing this array contains row indices, such that $pntrr(i) - pntrb(1)$ is the last index of block row *i* in the array *indx*.

For zero-based indexing this array contains row indices, such that $pntrr(i) - pntrb(0) - 1$ is the last index of block row *i* in the array *indx*.

Refer to *pointerE* array description in [BSR Format](#) for more details.

x

REAL for `mkl_sbsrsv`.

DOUBLE PRECISION for `mkl_dbsrsv`.

COMPLEX for `mkl_cbsrsv`.

DOUBLE COMPLEX for `mkl_zbsrsv`.

Array, size at least $(m \cdot lb)$.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL for `mkl_sbsrsv`.

DOUBLE PRECISION for `mkl_dbsrsv`.

COMPLEX for `mkl_cbsrsv`.

DOUBLE COMPLEX for `mkl_zbsrsv`.

Array, size at least $(m \cdot lb)$.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha
```

```
REAL val(*)
```

```
REAL x(*), y(*)
```

```
SUBROUTINE mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*)
```

```
DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha
```

```
COMPLEX val(*)
```

```
COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lb
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*)
```

```
DOUBLE COMPLEX x(*), y(*)
```

mkl_?cscsv

Solves a system of linear equations for a sparse matrix in the CSC format.

Syntax

```
call mkl_scscsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_ccscsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
call mkl_zcscsv(transa, m, alpha, matdescra, val, indx, pntreb, pntre, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?cscsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A') \cdot x$,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scscsv</code> . DOUBLE PRECISION for <code>mkl_dcscsv</code> . COMPLEX for <code>mkl_ccscsv</code> . DOUBLE COMPLEX for <code>mkl_zcscsv</code> . Specifies the scalar <i>alpha</i> .

matdescra CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val REAL for `mkl_scscsv`.
 DOUBLE PRECISION for `mkl_dcscsv`.
 COMPLEX for `mkl_ccscsv`.
 DOUBLE COMPLEX for `mkl_zcscsv`.
 Array containing non-zero elements of the matrix *A*.
 For one-based indexing its length is $pntrb(m) - pntrb(1)$.
 For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.
 Refer to *values* array description in [CSC Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).
 No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the row indices for each non-zero element of the matrix *A*.
 Its length is equal to length of the *val* array.
 Refer to *columns* array description in [CSC Format](#) for more details.

NOTE

Row indices must be sorted in increasing order for each column.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column *i* in the arrays *val* and *indx*.
 Refer to *pointerb* array description in [CSC Format](#) for more details.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1)$ is the last index of column *i* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) - 1$ is the last index of column i in the arrays val and $indx$.

Refer to *pointerE* array description in [CSC Format](#) for more details.

x

REAL for `mkl_scscsv`.
 DOUBLE PRECISION for `mkl_dcscsv`.
 COMPLEX for `mkl_ccscsv`.
 DOUBLE COMPLEX for `mkl_zcscsv`.

Array, size at least m .

On entry, the array x must contain the vector x . The elements are accessed with unit increment.

y

REAL for `mkl_scscsv`.
 DOUBLE PRECISION for `mkl_dcscsv`.
 COMPLEX for `mkl_ccscsv`.
 DOUBLE COMPLEX for `mkl_zcscsv`.

Array, size at least m .

On entry, the array y must contain the vector y . The elements are accessed with unit increment.

Output Parameters

y

Contains the solution vector x .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntrb, pntrb, x, y)
```

```
CHARACTER*1  transa
```

```
CHARACTER    matdescra(*)
```

```
INTEGER      m
```

```
INTEGER      indx(*), pntrb(m), pntrb(m)
```

```
REAL         alpha
```

```
REAL         val(*)
```

```
REAL         x(*), y(*)
```

SUBROUTINE `mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)`

CHARACTER*1 `transa`

CHARACTER `matdescra(*)`

INTEGER `m`

INTEGER `indx(*)`, `pntrb(m)`, `pntre(m)`

DOUBLE PRECISION `alpha`

DOUBLE PRECISION `val(*)`

DOUBLE PRECISION `x(*)`, `y(*)`

SUBROUTINE `mkl_ccscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)`

CHARACTER*1 `transa`

CHARACTER `matdescra(*)`

INTEGER `m`

INTEGER `indx(*)`, `pntrb(m)`, `pntre(m)`

COMPLEX `alpha`

COMPLEX `val(*)`

COMPLEX `x(*)`, `y(*)`

SUBROUTINE `mkl_zcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)`

CHARACTER*1 `transa`

CHARACTER `matdescra(*)`

INTEGER `m`

INTEGER `indx(*)`, `pntrb(m)`, `pntre(m)`

DOUBLE COMPLEX `alpha`

DOUBLE COMPLEX `val(*)`

DOUBLE COMPLEX `x(*)`, `y(*)`

mkl_?coosv

Solves a system of linear equations for a sparse matrix in the coordinate format.

Syntax

`call mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)`

`call mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)`

`call mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)`

`call mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)`

Include Files

- `mkl.fi`

Description

The `mkl_?coosv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A') \cdot x$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scoosv</code> . DOUBLE PRECISION for <code>mkl_dcoosv</code> . COMPLEX for <code>mkl_ccoosv</code> . DOUBLE COMPLEX for <code>mkl_zcoosv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scoosv</code> . DOUBLE PRECISION for <code>mkl_dcoosv</code> . COMPLEX for <code>mkl_ccoosv</code> . DOUBLE COMPLEX for <code>mkl_zcoosv</code> . Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.

<i>rowind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>INTEGER. Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>INTEGER. Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>REAL for <code>mkl_scoosv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoosv</code>.</p> <p>COMPLEX for <code>mkl_ccoosv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoosv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.</p>
<i>y</i>	<p>REAL for <code>mkl_scoosv</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcoosv</code>.</p> <p>COMPLEX for <code>mkl_ccoosv</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcoosv</code>.</p> <p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>y</i> must contain the vector <i>y</i>. The elements are accessed with unit increment.</p>

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .
----------	-------------------------------------

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1  transa
```

```
CHARACTER    matdescra(*)
```

```
INTEGER      m, nnz
```

```
INTEGER      rowind(*), colind(*)
```

```
REAL         alpha
```

```
REAL         val(*)
```

```
REAL         x(*), y(*)
```

```
SUBROUTINE mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*)
```

```
DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX alpha
```

```
COMPLEX val(*)
```

```
COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*)
```

```
DOUBLE COMPLEX x(*), y(*)
```

mkl_?csrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format.

Syntax

```
call mkl_scsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntbr, pntre, b, ldb, beta, c, ldc)
```

```
call mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntbr, pntre, b, ldb, beta, c, ldc)
```

```
call mkl_ccsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntbr, pntre, b, ldb, beta, c, ldc)
```

```
call mkl_zcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntbr, pntre, b, ldb, beta, c, ldc)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row (CSR) format, *A'* is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C,$</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for <code>mkl_scsrmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmm</code>.</p> <p>COMPLEX for <code>mkl_ccsrmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrmm</code>.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for <code>mkl_scsrmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmm</code>.</p>

	<p>COMPLEX for <code>mkl_ccsrmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrmm</code>.</p> <p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>For one-based indexing its length is $pntrb(m) - pntrb(1)$.</p> <p>For zero-based indexing its length is $pntrb(-1) - pntrb(0)$.</p> <p>Refer to <i>values</i> array description in CSR Format for more details.</p>
<i>indx</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>columns</i> array description in CSR Format for more details.</p>
<i>pntrb</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that $pntrb(I) - pntrb(1) + 1$ is the first index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that $pntrb(I) - pntrb(0)$ is the first index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSR Format for more details.</p>
<i>pntrb</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that $pntrb(I) - pntrb(1)$ is the last index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that $pntrb(I) - pntrb(0) - 1$ is the last index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>b</i>	<p>REAL for <code>mkl_scsrmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmm</code>.</p> <p>COMPLEX for <code>mkl_ccsrmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrmm</code>.</p> <p>Array, size <i>ldb</i> by at least <i>n</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed for one-based indexing, and (at least <i>k</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed, <i>ldb</i>) for zero-based indexing.</p> <p>On entry with <i>transa</i>='N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.</p>
<i>beta</i>	<p>REAL for <code>mkl_scsrmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmm</code>.</p>

COMPLEX for `mkl_ccsrm`.

DOUBLE COMPLEX for `mkl_zcsrm`.

Specifies the scalar *beta*.

c REAL for `mkl_scsrm`.

DOUBLE PRECISION for `mkl_dcsrm`.

COMPLEX for `mkl_ccsrm`.

DOUBLE COMPLEX for `mkl_zcsrm`.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix ($\alpha * A * B + \beta * C$) or ($\alpha * A' * B + \beta * C$).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, k, ldb, ldc
INTEGER      indx(*), pntrb(m), pntre(m)
COMPLEX      alpha, beta
COMPLEX      val(*), b(ldb,*), c(ldc,*)

```

```

SUBROUTINE mkl_zcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, k, ldb, ldc
INTEGER      indx(*), pntrb(m), pntre(m)
DOUBLE COMPLEX alpha, beta
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)

```

mkl_?bsrmm

Computes matrix - matrix product of a sparse matrix stored in the BSR format.

Syntax

```

call mkl_sbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)
call mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)
call mkl_cbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)
call mkl_zbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)

```

Include Files

- mkl.fi

Description

The `mkl_?bsrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and C are dense matrices, A is an m -by- k sparse matrix in block sparse row (BSR) format, A' is the transpose of A .

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * A * B + \beta * C$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * A' * B + \beta * C$,</p>
<i>m</i>	INTEGER. Number of block rows of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix C .
<i>k</i>	INTEGER. Number of block columns of the matrix A .
<i>lb</i>	INTEGER. Size of the block in the matrix A .
<i>alpha</i>	<p>REAL for mkl_sbsrmm.</p> <p>DOUBLE PRECISION for mkl_dbsrmm.</p> <p>COMPLEX for mkl_cbsrmm.</p> <p>DOUBLE COMPLEX for mkl_zbsrmm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_sbsrmm.</p> <p>DOUBLE PRECISION for mkl_dbsrmm.</p> <p>COMPLEX for mkl_cbsrmm.</p> <p>DOUBLE COMPLEX for mkl_zbsrmm.</p> <p>Array containing elements of non-zero blocks of the matrix A. Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.</p>
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks in the matrix A . Refer to the `columns` array description in [BSR Format](#) for more details.

`pntrb`

INTEGER. Array of length m .

For one-based indexing: this array contains row indices, such that $pntrb(I) - pntrb(1) + 1$ is the first index of block row I in the array `indx`.

For zero-based indexing: this array contains row indices, such that $pntrb(I) - pntrb(0)$ is the first index of block row I in the array `indx`.

Refer to `pointerB` array description in [BSR Format](#) for more details.

`pntrr`

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrr(I) - pntrb(1)$ is the last index of block row I in the array `indx`.

For zero-based indexing this array contains row indices, such that $pntrr(I) - pntrb(0) - 1$ is the last index of block row I in the array `indx`.

Refer to `pointerE` array description in [BSR Format](#) for more details.

`b`

REAL for `mkl_sbsrmm`.

DOUBLE PRECISION for `mkl_dbsrmm`.

COMPLEX for `mkl_cbsrmm`.

DOUBLE COMPLEX for `mkl_zbsrmm`.

Array, size `ldb` by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, `ldb`) for zero-based indexing.

On entry with `transa='N'` or `'n'`, the leading n -by- k block part of the array `b` must contain the matrix B , otherwise the leading m -by- n block part of the array `b` must contain the matrix B .

`ldb`

INTEGER. Specifies the leading dimension (in blocks) of `b` as declared in the calling (sub)program.

`beta`

REAL for `mkl_sbsrmm`.

DOUBLE PRECISION for `mkl_dbsrmm`.

COMPLEX for `mkl_cbsrmm`.

DOUBLE COMPLEX for `mkl_zbsrmm`.

Specifies the scalar `beta`.

`c`

REAL for `mkl_sbsrmm`.

DOUBLE PRECISION for `mkl_dbsrmm`.

COMPLEX for `mkl_cbsrmm`.

DOUBLE COMPLEX for `mkl_zbsrmm`.

Array, size (`ldc`, n) for one-based indexing, size (k , `ldc`) for zero-based indexing.

On entry, the leading m -by- n block part of the array c must contain the matrix C , otherwise the leading n -by- k block part of the array c must contain the matrix C .

ldc INTEGER. Specifies the leading dimension (in blocks) of c as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
```

```
indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ld, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
```

```
indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ld, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_cbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
```

```
indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ld, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_zbsrmm(transa, m, n, k, lb, alpha, matdescra, val,
indx, pntreb, pntre, b, ldb, beta, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, k, ld, ldb, ldc
INTEGER      indx(*), pntreb(m), pntre(m)
DOUBLE COMPLEX  alpha, beta
DOUBLE COMPLEX  val(*), b(ldb,*), c(ldc,*)

```

mkl_?cscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format.

Syntax

```

call mkl_scscmm(transa, m, n, k, alpha, matdescra, val, indx, pntreb, pntre, b, ldb,
beta, c, ldc)
call mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntreb, pntre, b, ldb,
beta, c, ldc)
call mkl_ccscmm(transa, m, n, k, alpha, matdescra, val, indx, pntreb, pntre, b, ldb,
beta, c, ldc)
call mkl_zcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntreb, pntre, b, ldb,
beta, c, ldc)

```

Include Files

- mkl.fi

Description

The `mkl_?cscmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta * C$$

or

$$C := \alpha A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A'* is the transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for <code>mkl_scscmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscmm</code>.</p> <p>COMPLEX for <code>mkl_ccscmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscmm</code>.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for <code>mkl_scscmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscmm</code>.</p> <p>COMPLEX for <code>mkl_ccscmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscmm</code>.</p> <p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>For one-based indexing its length is $pntrb(k) - pntrb(1)$.</p> <p>For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.</p> <p>Refer to <i>values</i> array description in CSC Format for more details.</p>
<i>indx</i>	<p>INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>rows</i> array description in CSC Format for more details.</p>
<i>pntrb</i>	<p>INTEGER. Array of length <i>k</i>.</p> <p>For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSC Format for more details.</p>

<i>pntrc</i>	<p>INTEGER. Array of length k.</p> <p>For one-based indexing this array contains column indices, such that $pntrc(i) - pntrb(1)$ is the last index of column i in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains column indices, such that $pntrc(i) - pntrb(1) - 1$ is the last index of column i in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSC Format for more details.</p>
<i>b</i>	<p>REAL for <code>mkl_scscmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscmm</code>.</p> <p>COMPLEX for <code>mkl_ccscmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscmm</code>.</p> <p>Array, size <i>ldb</i> by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, <i>ldb</i>) for zero-based indexing.</p> <p>On entry with <i>transa</i> = 'N' or 'n', the leading k-by-n part of the array <i>b</i> must contain the matrix B, otherwise the leading m-by-n part of the array <i>b</i> must contain the matrix B.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.</p>
<i>beta</i>	<p>REAL*8. Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>REAL for <code>mkl_scscmm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscmm</code>.</p> <p>COMPLEX for <code>mkl_ccscmm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscmm</code>.</p> <p>Array, size <i>ldc</i> by n for one-based indexing, and (m, <i>ldc</i>) for zero-based indexing.</p> <p>On entry, the leading m-by-n part of the array <i>c</i> must contain the matrix C, otherwise the leading k-by-n part of the array <i>c</i> must contain the matrix C.</p>
<i>ldc</i>	<p>INTEGER. Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.</p>

Output Parameters

<i>c</i>	<p>Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.</p>
----------	---

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscmm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccscmm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcscmm(transa, m, n, k, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?coomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format.

Syntax

```
call mkl_scoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)
```

```
call mkl_dcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)
```

```
call mkl_ccoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)
```

```
call mkl_zcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?coomm` routine performs a matrix-matrix operation defined as

```
 $C := \alpha * A * B + \beta * C$ 
```

or

```
 $C := \alpha * A' * B + \beta * C,$ 
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the coordinate format, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C,$
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scoomm</code> .

DOUBLE PRECISION for `mkl_dcoomm`.

COMPLEX for `mkl_ccoomm`.

DOUBLE COMPLEX for `mkl_zcoomm`.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val

REAL for `mkl_scoomm`.

DOUBLE PRECISION for `mkl_dcoomm`.

COMPLEX for `mkl_ccoomm`.

DOUBLE COMPLEX for `mkl_zcoomm`.

Array of length *nnz*, contains non-zero elements of the matrix *A* in the arbitrary order.

Refer to *values* array description in [Coordinate Format](#) for more details.

rowind

INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*.

Refer to *columns* array description in [Coordinate Format](#) for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in [Coordinate Format](#) for more details.

b

REAL for `mkl_scoomm`.

DOUBLE PRECISION for `mkl_dcoomm`.

COMPLEX for `mkl_ccoomm`.

DOUBLE COMPLEX for `mkl_zcoomm`.

Array, size *ldb* by at least *n* for non-transposed matrix *A* and at least *m* for transposed for one-based indexing, and (at least *k* for non-transposed matrix *A* and at least *m* for transposed, *ldb*) for zero-based indexing.

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

beta

REAL for `mkl_scoomm`.

DOUBLE PRECISION for `mkl_dcoomm`.

COMPLEX for `mkl_ccoomm`.

DOUBLE COMPLEX for `mkl_zcoomm`.

Specifies the scalar *beta*.

c REAL for `mkl_scoomm`.

DOUBLE PRECISION for `mkl_dcoomm`.

COMPLEX for `mkl_ccoomm`.

DOUBLE COMPLEX for `mkl_zcoomm`.

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomm(transa, m, n, k, alpha, matdescra, val,
```

```
rowind, colind, nnz, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcoomm(transa, m, n, k, alpha, matdescra, val,
```

```
rowind, colind, nnz, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccoomm(transa, m, n, k, alpha, matdescra, val,  
rowind, colind, nnz, b, ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc, nnz  
INTEGER rowind(*), colind(*)  
COMPLEX alpha, beta  
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcoomm(transa, m, n, k, alpha, matdescra, val,  
rowind, colind, nnz, b, ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc, nnz  
INTEGER rowind(*), colind(*)  
DOUBLE COMPLEX alpha, beta  
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?csrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format.

Syntax

```
call mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c,  
ldc)  
call mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c,  
ldc)  
call mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c,  
ldc)  
call mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c,  
ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?csrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

```
 $C := \alpha \cdot \text{inv}(A) \cdot B$ 
```

or

```
 $C := \alpha \cdot \text{inv}(A') \cdot B,$ 
```

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the system of linear equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$,</p>
<i>m</i>	INTEGER. Number of columns of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix C .
<i>alpha</i>	<p>REAL for mkl_scsrsm.</p> <p>DOUBLE PRECISION for mkl_dcsrsm.</p> <p>COMPLEX for mkl_ccsrsm.</p> <p>DOUBLE COMPLEX for mkl_zcsrsm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_scsrsm.</p> <p>DOUBLE PRECISION for mkl_dcsrsm.</p> <p>COMPLEX for mkl_ccsrsm.</p> <p>DOUBLE COMPLEX for mkl_zcsrsm.</p> <p>Array containing non-zero elements of the matrix A.</p> <p>For one-based indexing its length is $\text{pntrc}(m) - \text{pntrb}(1)$.</p> <p>For zero-based indexing its length is $\text{pntrc}(m-1) - \text{pntrb}(0)$.</p> <p>Refer to <i>values</i> array description in CSR Format for more details.</p>

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.
 Its length is equal to length of the *val* array.
 Refer to *columns* array description in [CSR Format](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row *i* in the arrays *val* and *indx*.
 Refer to *pointerb* array description in [CSR Format](#) for more details.

pntrE INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntrE(i) - pntrb(1)$ is the last index of row *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains row indices, such that $pntrE(i) - pntrb(0) - 1$ is the last index of row *i* in the arrays *val* and *indx*.
 Refer to *pointerE* array description in [CSR Format](#) for more details.

b REAL for `mkl_scsrsm`.
 DOUBLE PRECISION for `mkl_dcsrsm`.
 COMPLEX for `mkl_ccsrsm`.
 DOUBLE COMPLEX for `mkl_zcsrsm`.
 Array, size (*ldb*, *n*) for one-based indexing, and (*m*, *ldb*) for zero-based indexing.
 On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL*8.
 Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.
 The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, n, ldb, ldc
INTEGER indx(*), pntrb(m), pntre(m)
DOUBLE COMPLEX alpha
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?cscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Syntax

```
call mkl_scscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
call mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
call mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
call mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?cscsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

```
 $C := \alpha * \text{inv}(A) * B$ 
```

or

```
 $C := \alpha * \text{inv}(A') * B,$ 
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the system of equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$,</p>
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	<p>REAL for <code>mkl_scscsm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscsm</code>.</p> <p>COMPLEX for <code>mkl_ccscsm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscsm</code>.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for <code>mkl_scscsm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcscsm</code>.</p> <p>COMPLEX for <code>mkl_ccscsm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcscsm</code>.</p> <p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>For one-based indexing its length is $\text{pntrb}(k) - \text{pntrb}(1)$.</p> <p>For zero-based indexing its length is $\text{pntrb}(m-1) - \text{pntrb}(0)$.</p> <p>Refer to <i>values</i> array description in CSC Format for more details.</p> <hr/> <p>NOTE</p> <p>The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).</p> <p>No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.</p> <hr/>
<i>indx</i>	<p>INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i>. Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>rows</i> array description in CSC Format for more details.</p> <hr/> <p>NOTE</p> <p>Row indices must be sorted in increasing order for each column.</p> <hr/>
<i>pntrb</i>	INTEGER. Array of length <i>m</i> .

For one-based indexing this array contains column indices, such that $pntrb(I) - pntrb(1) + 1$ is the first index of column I in the arrays val and $indx$.

For zero-based indexing this array contains column indices, such that $pntrb(I) - pntrb(0)$ is the first index of column I in the arrays val and $indx$.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntrc

INTEGER. Array of length m .

For one-based indexing this array contains column indices, such that $pntrc(I) - pntrb(1)$ is the last index of column I in the arrays val and $indx$.

For zero-based indexing this array contains column indices, such that $pntrc(I) - pntrb(1) - 1$ is the last index of column I in the arrays val and $indx$.

Refer to *pointerE* array description in [CSC Format](#) for more details.

b

REAL for `mkl_scscsm`.

DOUBLE PRECISION for `mkl_dcscsm`.

COMPLEX for `mkl_ccscsm`.

DOUBLE COMPLEX for `mkl_zcscsm`.

Array, size ldb by n for one-based indexing, and (m, ldb) for zero-based indexing.

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

REAL for `mkl_scscsm`.

DOUBLE PRECISION for `mkl_dcscsm`.

COMPLEX for `mkl_ccscsm`.

DOUBLE COMPLEX for `mkl_zcscsm`.

Array, size ldc by n for one-based indexing, and (m, ldc) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx,
```

```
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?coosm

Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Syntax

```
call mkl_scoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
call mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
call mkl_ccoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
call mkl_zcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?coosm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha \cdot \text{inv}(A) \cdot B$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha \cdot \text{inv}(A') \cdot B$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL for <code>mkl_scoosm</code> .

	DOUBLE PRECISION for <code>mkl_dcoosm</code> .
	COMPLEX for <code>mkl_ccoosm</code> .
	DOUBLE COMPLEX for <code>mkl_zcoosm</code> .
	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scoosm</code> . DOUBLE PRECISION for <code>mkl_dcoosm</code> . COMPLEX for <code>mkl_ccoosm</code> . DOUBLE COMPLEX for <code>mkl_zcoosm</code> . Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>b</i>	REAL for <code>mkl_scoosm</code> . DOUBLE PRECISION for <code>mkl_dcoosm</code> . COMPLEX for <code>mkl_ccoosm</code> . DOUBLE COMPLEX for <code>mkl_zcoosm</code> . Array, size <i>ldb</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldb</i>) for zero-based indexing. Before entry the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL for `mkl_scoosm`.
 DOUBLE PRECISION for `mkl_dcoosm`.
 COMPLEX for `mkl_ccoosm`.
 DOUBLE COMPLEX for `mkl_zcoosm`.
 Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.
 The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
COMPLEX alpha
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, nnz
```

```
INTEGER rowind(*), colind(*)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?bsrsm

Solves a system of linear matrix equations for a sparse matrix in the BSR format.

Syntax

```
call mkl_scsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

```
call mkl_dcsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

```
call mkl_ccsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

```
call mkl_zcsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?bsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * \text{inv}(A) * B$.</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * \text{inv}(A') * B$.</p>
<i>m</i>	INTEGER. Number of block columns of the matrix A.
<i>n</i>	INTEGER. Number of columns of the matrix C.
<i>lb</i>	INTEGER. Size of the block in the matrix A.
<i>alpha</i>	<p>REAL for mkl_sbsrsm.</p> <p>DOUBLE PRECISION for mkl_dbsrsm.</p> <p>COMPLEX for mkl_cbsrsm.</p> <p>DOUBLE COMPLEX for mkl_zbsrsm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_sbsrsm.</p> <p>DOUBLE PRECISION for mkl_dbsrsm.</p> <p>COMPLEX for mkl_cbsrsm.</p> <p>DOUBLE COMPLEX for mkl_zbsrsm.</p> <p>Array containing elements of non-zero blocks of the matrix A. Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.</p>
<hr/> <p>NOTE</p> <p>The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).</p> <p>No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.</p> <hr/>	
<i>indx</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the number of non-zero blocks in the matrix A.</p> <p>Refer to the <i>columns</i> array description in BSR Format for more details.</p>
<i>pntrb</i>	INTEGER. Array of length <i>m</i> .

For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row i in the array $indx$.

For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row i in the array $indx$.

Refer to *pointerB* array description in [BSR Format](#) for more details.

pntrE

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrE(i) - pntrb(1)$ is the last index of block row i in the array $indx$.

For zero-based indexing this array contains row indices, such that $pntrE(i) - pntrb(0) - 1$ is the last index of block row i in the array $indx$.

Refer to *pointerE* array description in [BSR Format](#) for more details.

b

REAL for `mkl_sbsrsm`.

DOUBLE PRECISION for `mkl_dbsrsm`.

COMPLEX for `mkl_cbsrsm`.

DOUBLE COMPLEX for `mkl_zbsrsm`.

Array, size (ldb, n) for one-based indexing, size (m, ldb) for zero-based indexing.

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension (in blocks) of b as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension (in blocks) of c as declared in the calling (sub)program.

Output Parameters

c

REAL for `mkl_sbsrsm`.

DOUBLE PRECISION for `mkl_dbsrsm`.

COMPLEX for `mkl_cbsrsm`.

DOUBLE COMPLEX for `mkl_zbsrsm`.

Array, size (ldc, n) for one-based indexing, size (m, ldc) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

SUBROUTINE mkl_sbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, lb, ldb, ldc

INTEGER indx(*), pntreb(m), pntre(m)

REAL alpha

REAL val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, lb, ldb, ldc

INTEGER indx(*), pntreb(m), pntre(m)

DOUBLE PRECISION alpha

DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, lb, ldb, ldc

INTEGER indx(*), pntreb(m), pntre(m)

COMPLEX alpha

COMPLEX val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntreb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, lb, ldb, ldc

INTEGER indx(*), pntreb(m), pntre(m)

DOUBLE COMPLEX alpha

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)

mkl_?diamv

Computes matrix - vector product for a sparse matrix in the diagonal format with one-based indexing.

Syntax

call mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag, ndiag, x, beta, y)

```
call mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
call mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
call mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, idiag, ndiag, x, beta, y)
```

Include Files

- mkl.fi

Description

The `mkl_?diamv` routine performs a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y$$

or

$$y := \alpha * A' * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored in the diagonal format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A' * x + \beta * y$.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sdiamv. DOUBLE PRECISION for mkl_ddiamv. COMPLEX for mkl_cdiamv. DOUBLE COMPLEX for mkl_zdiamv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

<i>val</i>	<p>REAL for <code>mkl_sdiamv</code>. DOUBLE PRECISION for <code>mkl_ddiamv</code>. COMPLEX for <code>mkl_cdiamv</code>. DOUBLE COMPLEX for <code>mkl_zdiamv</code>.</p> <p>Two-dimensional array of size <i>lval</i> by <i>ndiag</i>, contains non-zero diagonals of the matrix <i>A</i>. Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.</p>
<i>lval</i>	<p>INTEGER. Leading dimension of <i>val</i>, $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.</p>
<i>idiag</i>	<p>INTEGER. Array of length <i>ndiag</i>, contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i>.</p> <p>Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.</p>
<i>ndiag</i>	<p>INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i>.</p>
<i>x</i>	<p>REAL for <code>mkl_sdiamv</code>. DOUBLE PRECISION for <code>mkl_ddiamv</code>. COMPLEX for <code>mkl_cdiamv</code>. DOUBLE COMPLEX for <code>mkl_zdiamv</code>.</p> <p>Array, size at least <i>k</i> if <i>transa</i> = 'N' or 'n', and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>
<i>beta</i>	<p>REAL for <code>mkl_sdiamv</code>. DOUBLE PRECISION for <code>mkl_ddiamv</code>. COMPLEX for <code>mkl_cdiamv</code>. DOUBLE COMPLEX for <code>mkl_zdiamv</code>.</p> <p>Specifies the scalar <i>beta</i>.</p>
<i>y</i>	<p>REAL for <code>mkl_sdiamv</code>. DOUBLE PRECISION for <code>mkl_ddiamv</code>. COMPLEX for <code>mkl_cdiamv</code>. DOUBLE COMPLEX for <code>mkl_zdiamv</code>.</p> <p>Array, size at least <i>m</i> if <i>transa</i> = 'N' or 'n', and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i>.</p>

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL alpha, beta
```

```
REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lval, ndiag
```

```
INTEGER idiag(*)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, idiag,
ndiag, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

mkl_?skymv

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing.

Syntax

```
call mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
call mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
call mkl_cskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
call mkl_zskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

Include Files

- mkl.fi

Description

The mkl_?skymv routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored using the skyline storage scheme, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sskymv. DOUBLE PRECISION for mkl_dskymv. COMPLEX for mkl_cskymv. DOUBLE COMPLEX for mkl_zskymv.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

NOTE

General matrices (*matdescra*(1)='G') is not supported.

val

REAL for `mkl_sskymv`.

DOUBLE PRECISION for `mkl_dskymv`.

COMPLEX for `mkl_cskymv`.

DOUBLE COMPLEX for `mkl_zskymv`.

Array containing the set of elements of the matrix *A* in the skyline profile form.

If *matdescrsa*(2) = 'L', then *val* contains elements from the low triangle of the matrix *A*.

If *matdescrsa*(2) = 'U', then *val* contains elements from the upper triangle of the matrix *A*.

Refer to *values* array description in [Skyline Storage Scheme](#) for more details.

pntr

INTEGER. Array of length (*m* + 1) for lower triangle, and (*k* + 1) for upper triangle.

It contains the indices specifying in the *val* the positions of the first element in each row (column) of the matrix *A*. Refer to *pointers* array description in [Skyline Storage Scheme](#) for more details.

x

REAL for `mkl_sskymv`.

DOUBLE PRECISION for `mkl_dskymv`.

COMPLEX for `mkl_cskymv`.

DOUBLE COMPLEX for `mkl_zskymv`.

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for `mkl_sskymv`.

DOUBLE PRECISION for `mkl_dskymv`.

COMPLEX for `mkl_cskymv`.

DOUBLE COMPLEX for `mkl_zskymv`.

Specifies the scalar *beta*.

y

REAL for `mkl_sskymv`.

DOUBLE PRECISION for `mkl_dskymv`.

COMPLEX for `mkl_cskymv`.

DOUBLE COMPLEX for `mkl_zskymv`.

Array, size at least m if `transa = 'N'` or `'n'` and at least k otherwise. On entry, the array `y` must contain the vector `y`.

Output Parameters

`y` Overwritten by the updated vector `y`.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
CHARACTER*1  transa
```

```
CHARACTER    matdescra(*)
```

```
INTEGER      m, k
```

```
INTEGER      pntr(*)
```

```
REAL         alpha, beta
```

```
REAL         val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
CHARACTER*1  transa
```

```
CHARACTER    matdescra(*)
```

```
INTEGER      m, k
```

```
INTEGER      pntr(*)
```

```
DOUBLE PRECISION  alpha, beta
```

```
DOUBLE PRECISION  val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
CHARACTER*1  transa
```

```
CHARACTER    matdescra(*)
```

```
INTEGER      m, k
```

```
INTEGER      pntr(*)
```

```
COMPLEX      alpha, beta
```

```
COMPLEX      val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zskymv(transa, m, k, alpha, matdescra, val, pnt, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER pnt(*)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?diasv

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

```
call mkl_ddiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

```
call mkl_cdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

```
call mkl_zdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

Include Files

- mkl.fi

Description

The `mkl_?diasv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

```
 $y := \alpha \cdot \text{inv}(A) \cdot x$ 
```

or

```
 $y := \alpha \cdot \text{inv}(A') \cdot x,$ 
```

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

transa CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$

	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * \text{inv}(A') * x$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_sdiasv</code> . DOUBLE PRECISION for <code>mkl_ddiasv</code> . COMPLEX for <code>mkl_cdiasv</code> . DOUBLE COMPLEX for <code>mkl_zdiasv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_sdiasv</code> . DOUBLE PRECISION for <code>mkl_ddiasv</code> . COMPLEX for <code>mkl_cdiasv</code> . DOUBLE COMPLEX for <code>mkl_zdiasv</code> . Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .
<hr/> NOTE <hr/>	
All elements of this array must be sorted in increasing order.	
<hr/>	
Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.	
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	REAL for <code>mkl_sdiasv</code> . DOUBLE PRECISION for <code>mkl_ddiasv</code> . COMPLEX for <code>mkl_cdiasv</code> . DOUBLE COMPLEX for <code>mkl_zdiasv</code> . Array, size at least <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> . The elements are accessed with unit increment.

y REAL for `mkl_sdiasv`.
 DOUBLE PRECISION for `mkl_ddiasv`.
 COMPLEX for `mkl_cdiasv`.
 DOUBLE COMPLEX for `mkl_zdiasv`.
 Array, size at least m .
 On entry, the array y must contain the vector y . The elements are accessed with unit increment.

Output Parameters

y Contains solution vector x .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL alpha
```

```
REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, lval, ndiag
```

```
INTEGER idiag(*)
```

```
COMPLEX alpha
```

```
COMPLEX val(lval,*), x(*), y(*)
```

SUBROUTINE `mkl_zdiasv(transa, m, alpha, matdescra, val, lval, idiag, ndiag, x, y)`

CHARACTER*1 `transa`

CHARACTER `matdescra(*)`

INTEGER `m, lval, ndiag`

INTEGER `idiag(*)`

DOUBLE COMPLEX `alpha`

DOUBLE COMPLEX `val(lval,*), x(*), y(*)`

mkl_?skysv

Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing.

Syntax

`call mkl_sskysv(transa, m, alpha, matdescra, val, pntr, x, y)`

`call mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)`

`call mkl_cskysv(transa, m, alpha, matdescra, val, pntr, x, y)`

`call mkl_zskysv(transa, m, alpha, matdescra, val, pntr, x, y)`

Include Files

- `mkl.fi`

Description

The `mkl_?skysv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

`y := alpha*inv(A)*x`

or

`y := alpha*inv(A')*x,`

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

transa CHARACTER*1. Specifies the system of linear equations.

If *transa* = 'N' or 'n', then `y := alpha*inv(A)*x`

If *transa* = 'T' or 't' or 'C' or 'c', then `y := alpha*inv(A')*x,`

<i>m</i>	<p>INTEGER. Number of rows of the matrix <i>A</i>.</p>
<i>alpha</i>	<p>REAL for mkl_sskysv. DOUBLE PRECISION for mkl_dskysv. COMPLEX for mkl_cskysv. DOUBLE COMPLEX for mkl_zskysv.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<hr/> <p>NOTE General matrices (<i>matdescra</i>(1)='G') is not supported.</p> <hr/>	
<i>val</i>	<p>REAL for mkl_sskysv. DOUBLE PRECISION for mkl_dskysv. COMPLEX for mkl_cskysv. DOUBLE COMPLEX for mkl_zskysv.</p> <p>Array containing the set of elements of the matrix <i>A</i> in the skyline profile form.</p> <p>If <i>matdescra</i>(2) = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i>.</p> <p>If <i>matdescra</i>(2) = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i>.</p> <p>Refer to <i>values</i> array description in Skyline Storage Scheme for more details.</p>
<i>pntr</i>	<p>INTEGER. Array of length (<i>m</i> + 1) for lower triangle, and (<i>k</i> + 1) for upper triangle.</p> <p>It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix <i>A</i>. Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.</p>
<i>x</i>	<p>REAL for mkl_sskysv. DOUBLE PRECISION for mkl_dskysv. COMPLEX for mkl_cskysv. DOUBLE COMPLEX for mkl_zskysv.</p> <p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.</p>

y REAL for mkl_sskysv.
 DOUBLE PRECISION for mkl_dskysv.
 COMPLEX for mkl_cskysv.
 DOUBLE COMPLEX for mkl_zskysv.
 Array, size at least m .
 On entry, the array y must contain the vector y . The elements are accessed with unit increment.

Output Parameters

y Contains solution vector x .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER pntr(*)
```

```
REAL alpha
```

```
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER pntr(*)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER pntr(*)
```

```
COMPLEX alpha
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zskysv(transa, m, alpha, matdescra, val, pntr, x, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m
```

```
INTEGER pntr(*)
```

```
DOUBLE COMPLEX alpha
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

mkl_?diamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb,
beta, c, ldc)
```

```
call mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb,
beta, c, ldc)
```

```
call mkl_cdiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb,
beta, c, ldc)
```

```
call mkl_zdiamm(transa, m, n, k, alpha, matdescra, val, lval, idiag, ndiag, b, ldb,
beta, c, ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?diamm` routine performs a matrix-matrix operation defined as

```
 $C := \alpha A * B + \beta C$ 
```

or

```
 $C := \alpha A' * B + \beta C,$ 
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the diagonal format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$,</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$.</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for <code>mkl_sdiamm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddiamm</code>.</p> <p>COMPLEX for <code>mkl_cdiamm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdiamm</code>.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for <code>mkl_sdiamm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddiamm</code>.</p> <p>COMPLEX for <code>mkl_cdiamm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdiamm</code>.</p> <p>Two-dimensional array of size <i>lval</i> by <i>ndiag</i>, contains non-zero diagonals of the matrix <i>A</i>. Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.</p>
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	<p>INTEGER. Array of length <i>ndiag</i>, contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i>.</p> <p>Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.</p>
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>b</i>	<p>REAL for <code>mkl_sdiamm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddiamm</code>.</p> <p>COMPLEX for <code>mkl_cdiamm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdiamm</code>.</p> <p>Array, size (<i>ldb</i>, <i>n</i>).</p>

On entry with $transa = 'N'$ or $'n'$, the leading k -by- n part of the array b must contain the matrix B , otherwise the leading m -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program.

beta REAL for mkl_sdiamm.
DOUBLE PRECISION for mkl_ddiamm.
COMPLEX for mkl_cdiamm.
DOUBLE COMPLEX for mkl_zdiamm.
Specifies the scalar β .

c REAL for mkl_sdiamm.
DOUBLE PRECISION for mkl_ddiamm.
COMPLEX for mkl_cdiamm.
DOUBLE COMPLEX for mkl_zdiamm.
Array, size ldc by n .
On entry, the leading m -by- n part of the array c must contain the matrix C , otherwise the leading k -by- n part of the array c must contain the matrix C .

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamm(transa, m, n, k, alpha, matdescra, val, lval,
idiag, ndiag, b, ldb, beta, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, k, ldb, ldc, lval, ndiag
INTEGER      idiag(*)
REAL         alpha, beta
REAL         val(lval,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval,  
idiag, ndiag, b, ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc, lval, ndiag  
INTEGER idiag(*)  
DOUBLE PRECISION alpha, beta  
DOUBLE PRECISION val(lval,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_cdiamm(transa, m, n, k, alpha, matdescra, val, lval,  
idiag, ndiag, b, ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc, lval, ndiag  
INTEGER idiag(*)  
COMPLEX alpha, beta  
COMPLEX val(lval,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zdiamm(transa, m, n, k, alpha, matdescra, val, lval,  
idiag, ndiag, b, ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc, lval, ndiag  
INTEGER idiag(*)  
DOUBLE COMPLEX alpha, beta  
DOUBLE COMPLEX val(lval,*), b(ldb,*), c(ldc,*)
```

mkl_?skymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing.

Syntax

```
call mkl_sskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)  
call mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)  
call mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)  
call mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pntr, b, ldb, beta, c, ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?sskymm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A' * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * A' * B + \beta * C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_sskymm</code> . DOUBLE PRECISION for <code>mkl_dskymm</code> . COMPLEX for <code>mkl_cskymm</code> . DOUBLE COMPLEX for <code>mkl_zskymm</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra* (1)='G') is not supported.

<i>val</i>	REAL for <code>mkl_sskymm</code> .
------------	------------------------------------

DOUBLE PRECISION for `mkl_dskymm`.

COMPLEX for `mkl_cskymm`.

DOUBLE COMPLEX for `mkl_zskymm`.

Array containing the set of elements of the matrix A in the skyline profile form.

If `matdescrsa(2) = 'L'`, then `val` contains elements from the low triangle of the matrix A .

If `matdescrsa(2) = 'U'`, then `val` contains elements from the upper triangle of the matrix A .

Refer to `values` array description in [Skyline Storage Scheme](#) for more details.

`pntr`

INTEGER. Array of length $(m + 1)$ for lower triangle, and $(k + 1)$ for upper triangle.

It contains the indices specifying the positions of the first element of the matrix A in each row (for the lower triangle) or column (for upper triangle) in the `val` array. Refer to `pointers` array description in [Skyline Storage Scheme](#) for more details.

`b`

REAL for `mkl_sskymm`.

DOUBLE PRECISION for `mkl_dskymm`.

COMPLEX for `mkl_cskymm`.

DOUBLE COMPLEX for `mkl_zskymm`.

Array, size (ldb, n) .

On entry with `transa = 'N'` or `'n'`, the leading k -by- n part of the array `b` must contain the matrix B , otherwise the leading m -by- n part of the array `b` must contain the matrix B .

`ldb`

INTEGER. Specifies the leading dimension of `b` as declared in the calling (sub)program.

`beta`

REAL for `mkl_sskymm`.

DOUBLE PRECISION for `mkl_dskymm`.

COMPLEX for `mkl_cskymm`.

DOUBLE COMPLEX for `mkl_zskymm`.

Specifies the scalar `beta`.

`c`

REAL for `mkl_sskymm`.

DOUBLE PRECISION for `mkl_dskymm`.

COMPLEX for `mkl_cskymm`.

DOUBLE COMPLEX for `mkl_zskymm`.

Array, size `ldc` by `n`.

On entry, the leading m -by- n part of the array `c` must contain the matrix C , otherwise the leading k -by- n part of the array `c` must contain the matrix C .

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
```

```
ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER pntr(*)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
```

```
ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER pntr(*)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
```

```
ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER pntr(*)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,  
ldb, beta, c, ldc)  
CHARACTER*1 transa  
CHARACTER matdescra(*)  
INTEGER m, n, k, ldb, ldc  
INTEGER pntr(*)  
DOUBLE COMPLEX alpha, beta  
DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

mkl_?diasm

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing.

Syntax

```
call mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c,  
ldc)  
call mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c,  
ldc)  
call mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c,  
ldc)  
call mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, idiag, ndiag, b, ldb, c,  
ldc)
```

Include Files

- mkl.fi

Description

The `mkl_?diasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the system of linear equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$,</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$.</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	<p>REAL for mkl_sdiasm.</p> <p>DOUBLE PRECISION for mkl_ddiasm.</p> <p>COMPLEX for mkl_cdiasm.</p> <p>DOUBLE COMPLEX for mkl_zdiasm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>REAL for mkl_sdiasm.</p> <p>DOUBLE PRECISION for mkl_ddiasm.</p> <p>COMPLEX for mkl_cdiasm.</p> <p>DOUBLE COMPLEX for mkl_zdiasm.</p> <p>Two-dimensional array of size <i>lval</i> by <i>ndiag</i>, contains non-zero diagonals of the matrix <i>A</i>. Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.</p>
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>b</i>	<p>REAL for mkl_sdiasm.</p> <p>DOUBLE PRECISION for mkl_ddiasm.</p> <p>COMPLEX for mkl_cdiasm.</p> <p>DOUBLE COMPLEX for mkl_zdiasm.</p>

Array, size (ldb, n) .

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program.

Output Parameters

c REAL for `mkl_sdiasm`.
 DOUBLE PRECISION for `mkl_ddiasm`.
 COMPLEX for `mkl_cdiasm`.
 DOUBLE COMPLEX for `mkl_zdiasm`.
 Array, size ldc by n .
 The leading m -by- n part of the array c contains the matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
```

```
ndiag, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL alpha
```

```
REAL val(lval,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
```

```
ndiag, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(lval,*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
ndiag, b, ldb, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, ldb, ldc, lval, ndiag
INTEGER      idiag(*)
COMPLEX      alpha
COMPLEX      val(lval,*), b(ldb,*), c(ldc,*)

```

```

SUBROUTINE mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, idiag,
ndiag, b, ldb, c, ldc)
CHARACTER*1  transa
CHARACTER    matdescra(*)
INTEGER      m, n, ldb, ldc, lval, ndiag
INTEGER      idiag(*)
DOUBLE COMPLEX alpha
DOUBLE COMPLEX val(lval,*), b(ldb,*), c(ldc,*)

```

mkl_?skysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing.

Syntax

```

call mkl_sskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_cskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
call mkl_zskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)

```

Include Files

- mkl.fi

Description

The `mkl_?skysm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A') * B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	<p>CHARACTER*1. Specifies the system of linear equations.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$,</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A') * B$,</p>
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	<p>REAL for mkl_sskysm.</p> <p>DOUBLE PRECISION for mkl_dskysm.</p> <p>COMPLEX for mkl_cskysm.</p> <p>DOUBLE COMPLEX for mkl_zskysm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	<p>CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<hr/> <p>NOTE</p> <p>General matrices (<i>matdescra</i>(1)='G') is not supported.</p> <hr/>	
<i>val</i>	<p>REAL for mkl_sskysm.</p> <p>DOUBLE PRECISION for mkl_dskysm.</p> <p>COMPLEX for mkl_cskysm.</p> <p>DOUBLE COMPLEX for mkl_zskysm.</p> <p>Array containing the set of elements of the matrix <i>A</i> in the skyline profile form.</p> <p>If <i>matdescrsa</i>(2) = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i>.</p> <p>If <i>matdescrsa</i>(2) = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i>.</p> <p>Refer to <i>values</i> array description in Skyline Storage Scheme for more details.</p>

<i>pntr</i>	<p>INTEGER. Array of length $(m + 1)$ for lower triangle, and $(n + 1)$ for upper triangle.</p> <p>It contains the indices specifying the positions of the first element of the matrix <i>A</i> in each row (for the lower triangle) or column (for upper triangle) in the <i>val</i> array. Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.</p>
<i>b</i>	<p>REAL for <code>mkl_sskysm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dskysm</code>.</p> <p>COMPLEX for <code>mkl_cskysm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zskysm</code>.</p> <p>Array, size (ldb, n).</p> <p>On entry the leading m-by-n part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.</p>
<i>ldc</i>	<p>INTEGER. Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.</p>

Output Parameters

<i>c</i>	<p>REAL for <code>mkl_sskysm</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dskysm</code>.</p> <p>COMPLEX for <code>mkl_cskysm</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zskysm</code>.</p> <p>Array, size ldc by n.</p> <p>The leading m-by-n part of the array <i>c</i> contains the matrix <i>C</i>.</p>
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER pntr(*)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

SUBROUTINE mkl_dskysm(transa, m, n, alpha, matdescra, val, pnttr, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER pnttr(*)

DOUBLE PRECISION alpha

DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cskysm(transa, m, n, alpha, matdescra, val, pnttr, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER pnttr(*)

COMPLEX alpha

COMPLEX val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zskysm(transa, m, n, alpha, matdescra, val, pnttr, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER pnttr(*)

DOUBLE COMPLEX alpha

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)

mkl_?dnscsr

Convert a sparse matrix in uncompressed representation to the CSR format and vice versa.

Syntax

call mkl_sdncsr(job, m, n, adns, lda, acsr, ja, ia, info)

call mkl_ddncsr(job, m, n, adns, lda, acsr, ja, ia, info)

call mkl_cdncsr(job, m, n, adns, lda, acsr, ja, ia, info)

call mkl_zdncsr(job, m, n, adns, lda, acsr, ja, ia, info)

Include Files

- mkl.fi

Description

This routine converts a sparse matrix A between formats: stored as a rectangular array (dense representation) and stored using compressed sparse row (CSR) format (3-array variation).

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	<p>INTEGER</p> <p>Array, contains the following conversion parameters:</p> <ul style="list-style-type: none"> • <i>job</i>(1): Conversion type. <ul style="list-style-type: none"> • If <i>job</i>(1)=0, the rectangular matrix <i>A</i> is converted to the CSR format; • if <i>job</i>(1)=1, the rectangular matrix <i>A</i> is restored from the CSR format. • <i>job</i>(2): index base for the rectangular matrix <i>A</i>. <ul style="list-style-type: none"> • If <i>job</i>(2)=0, zero-based indexing for the rectangular matrix <i>A</i> is used; • if <i>job</i>(2)=1, one-based indexing for the rectangular matrix <i>A</i> is used. • <i>job</i>(3): Index base for the matrix in CSR format. <ul style="list-style-type: none"> • If <i>job</i>(3)=0, zero-based indexing for the matrix in CSR format is used; • if <i>job</i>(3)=1, one-based indexing for the matrix in CSR format is used. • <i>job</i>(4): Portion of matrix. <ul style="list-style-type: none"> • If <i>job</i>(4)=0, <i>adns</i> is a lower triangular part of matrix <i>A</i>; • If <i>job</i>(4)=1, <i>adns</i> is an upper triangular part of matrix <i>A</i>; • If <i>job</i>(4)=2, <i>adns</i> is a whole matrix <i>A</i>. • <i>job</i>(5)=<i>nzmax</i>: maximum number of the non-zero elements allowed if <i>job</i>(1)=0. • <i>job</i>(6): job indicator for conversion to CSR format. <ul style="list-style-type: none"> • If <i>job</i>(6)=0, only array <i>ia</i> is generated for the output storage. • If <i>job</i>(6)>0, arrays <i>acsr</i>, <i>ia</i>, <i>ja</i> are generated for the output storage.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>adns</i>	<p>(input/output)</p> <p>REAL for <code>mkl_sdnscsr</code>.</p> <p>DOUBLE PRECISION for <code>mkl_ddnscsr</code>.</p> <p>COMPLEX for <code>mkl_cdnscsr</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zdnscsr</code>.</p> <p>If the conversion type is from uncompressed to CSR, on input <i>adns</i> contains an uncompressed (dense) representation of matrix <i>A</i>.</p>
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>adns</i> as declared in the calling (sub)program.

For zero-based indexing of A , lda must be at least $\max(1, n)$.

For one-based indexing of A , lda must be at least $\max(1, m)$.

acsr

(input/output)

REAL for `mkl_sdnscsr`.

DOUBLE PRECISION for `mkl_ddnscsr`.

COMPLEX for `mkl_cdnscsr`.

DOUBLE COMPLEX for `mkl_zdnscsr`.

If conversion type is from CSR to uncompressed, on input *acsr* contains the non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. If conversion type is from CSR to uncompressed, on input *ja* contains the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $m + 1$.

If conversion type is from CSR to uncompressed, on input *ia* contains indices of elements in the array *acsr*, such that $ia(i)$ is the index in the array *acsr* of the first non-zero element from the row i .

The value of $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

adns

If conversion type is from CSR to uncompressed, on output *adns* contains the uncompressed (dense) representation of matrix A .

acsr, ja, ia

If conversion type is from uncompressed to CSR, on output *acsr*, *ja*, and *ia* contain the compressed sparse row (CSR) format (3-array variation) of matrix A (see [Sparse Matrix Storage Formats](#) for a description of the storage format).

info

INTEGER. Integer info indicator only for restoring the matrix A from the CSR format.

If $info=0$, the execution is successful.

If $info=i$, the routine is interrupted processing the i -th row because there is no space in the arrays *acsr* and *ja* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
INTEGER      job(8)
```

```
INTEGER      m, n, lda, info
```

```
INTEGER      ja(*), ia(m+1)
```

```
REAL         adns(*), acsr(*)
```

```
SUBROUTINE mkl_ddnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
INTEGER      job(8)
```

```
INTEGER      m, n, lda, info
```

```
INTEGER      ja(*), ia(m+1)
```

```
DOUBLE PRECISION  adns(*), acsr(*)
```

```
SUBROUTINE mkl_cdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
INTEGER      job(8)
```

```
INTEGER      m, n, lda, info
```

```
INTEGER      ja(*), ia(m+1)
```

```
COMPLEX      adns(*), acsr(*)
```

```
SUBROUTINE mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
INTEGER      job(8)
```

```
INTEGER      m, n, lda, info
```

```
INTEGER      ja(*), ia(m+1)
```

```
DOUBLE COMPLEX  adns(*), acsr(*)
```

mkl_?csrcoo

Converts a sparse matrix in the CSR format to the coordinate format and vice versa.

Syntax

```
call mkl_scsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
call mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
call mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
call mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

Include Files

- mkl.fi

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to coordinate format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	<p>INTEGER</p> <p>Array, contains the following conversion parameters:</p> <p><i>job</i>(1)</p> <p>If <i>job</i>(1)=0, the matrix in the CSR format is converted to the coordinate format;</p> <p>if <i>job</i>(1)=1, the matrix in the coordinate format is converted to the CSR format.</p> <p>if <i>job</i>(1)=2, the matrix in the coordinate format is converted to the CSR format, and the column indices in CSR representation are sorted in the increasing order within each row.</p> <p><i>job</i>(2)</p> <p>If <i>job</i>(2)=0, zero-based indexing for the matrix in CSR format is used;</p> <p>if <i>job</i>(2)=1, one-based indexing for the matrix in CSR format is used.</p> <p><i>job</i>(3)</p> <p>If <i>job</i>(3)=0, zero-based indexing for the matrix in coordinate format is used;</p> <p>if <i>job</i>(3)=1, one-based indexing for the matrix in coordinate format is used.</p> <p><i>job</i>(5)</p> <p><i>job</i>(5)=<i>nzmax</i> - maximum number of the non-zero elements allowed if <i>job</i>(1)=0.</p> <p><i>job</i>(6) - job indicator.</p> <p>For conversion to the coordinate format:</p> <p>If <i>job</i>(6)=1, only array <i>rowind</i> is filled in for the output storage.</p> <p>If <i>job</i>(6)=2, arrays <i>rowind</i>, <i>colind</i> are filled in for the output storage.</p> <p>If <i>job</i>(6)=3, all arrays <i>rowind</i>, <i>colind</i>, <i>acoo</i> are filled in for the output storage.</p> <p>For conversion to the CSR format:</p> <p>If <i>job</i>(6)=0, all arrays <i>acsr</i>, <i>ja</i>, <i>ia</i> are filled in for the output storage.</p> <p>If <i>job</i>(6)=1, only array <i>ia</i> is filled in for the output storage.</p> <p>If <i>job</i>(6)=2, then it is assumed that the routine already has been called with the <i>job</i>(6)=1, and the user allocated the required space for storing the output arrays <i>acsr</i> and <i>ja</i>.</p>
<i>n</i>	INTEGER. Dimension of the matrix <i>A</i> .
<i>nnz</i>	INTEGER. Specifies the number of non-zero elements of the matrix <i>A</i> for <i>job</i> (1)≠0.

Refer to *nnz* description in [Coordinate Format](#) for more details.

acsr

(input/output)

REAL for `mkl_scsrcoo`.

DOUBLE PRECISION for `mkl_dcsrcoo`.

COMPLEX for `mkl_ccsrcoo`.

DOUBLE COMPLEX for `mkl_zcsrcoo`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $n + 1$, containing indices of elements in the array *acsr*, such that $ia(i)$ is the index in the array *acsr* of the first non-zero element from the row *i*. The value of the last element $ia(n + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

acoo

(input/output)

REAL for `mkl_scsrcoo`.

DOUBLE PRECISION for `mkl_dcsrcoo`.

COMPLEX for `mkl_ccsrcoo`.

DOUBLE COMPLEX for `mkl_zcsrcoo`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

rowind

(input/output) INTEGER. Array of length *nnz*, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

(input/output) INTEGER. Array of length *nnz*, contains the column indices for each non-zero element of the matrix *A*. Refer to *columns* array description in [Coordinate Format](#) for more details.

Output Parameters

nnz

Returns the number of converted elements of the matrix *A* for $job(1)=0$.

info

INTEGER. Integer info indicator only for converting the matrix *A* from the CSR format.

If $info=0$, the execution is successful.

If *info*=1, the routine is interrupted because there is no space in the arrays *acoo*, *rowind*, *colind* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
INTEGER job(8)
```

```
INTEGER n, nnz, info
```

```
INTEGER ja(*), ia(n+1), rowind(*), colind(*)
```

```
REAL acsr(*), acoo(*)
```

```
SUBROUTINE mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
INTEGER job(8)
```

```
INTEGER n, nnz, info
```

```
INTEGER ja(*), ia(n+1), rowind(*), colind(*)
```

```
DOUBLE PRECISION acsr(*), acoo(*)
```

```
SUBROUTINE mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
INTEGER job(8)
```

```
INTEGER n, nnz, info
```

```
INTEGER ja(*), ia(n+1), rowind(*), colind(*)
```

```
COMPLEX acsr(*), acoo(*)
```

```
SUBROUTINE mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

```
INTEGER job(8)
```

```
INTEGER n, nnz, info
```

```
INTEGER ja(*), ia(n+1), rowind(*), colind(*)
```

```
DOUBLE COMPLEX acsr(*), acoo(*)
```

mkl_?csrbsr

Converts a sparse matrix in the CSR format to the BSR format and vice versa.

Syntax

```
call mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
call mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
call mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
call mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

Include Files

- mkl.fi

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the block sparse row (BSR) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	<p>INTEGER</p> <p>Array, contains the following conversion parameters:</p> <p><i>job</i>(1)</p> <p>If <i>job</i>(1)=0, the matrix in the CSR format is converted to the BSR format; if <i>job</i>(1)=1, the matrix in the BSR format is converted to the CSR format.</p> <p><i>job</i>(2)</p> <p>If <i>job</i>(2)=0, zero-based indexing for the matrix in CSR format is used; if <i>job</i>(2)=1, one-based indexing for the matrix in CSR format is used.</p> <p><i>job</i>(3)</p> <p>If <i>job</i>(3)=0, zero-based indexing for the matrix in the BSR format is used; if <i>job</i>(3)=1, one-based indexing for the matrix in the BSR format is used.</p> <p><i>job</i>(4) is only used for conversion to CSR format. By default, the converter saves the blocks without checking whether an element is zero or not. If <i>job</i>(4)=1, then the converter only saves non-zero elements in blocks.</p> <p><i>job</i>(6) - job indicator.</p> <p>For conversion to the BSR format:</p> <p>If <i>job</i>(6)=0, only arrays <i>jab</i>, <i>iab</i> are generated for the output storage. If <i>job</i>(6)>0, all output arrays <i>absr</i>, <i>jab</i>, and <i>iab</i> are filled in for the output storage.</p> <p>If <i>job</i>(6)=-1, <i>iab</i>(1) returns the number of non-zero blocks.</p> <p>For conversion to the CSR format:</p> <p>If <i>job</i>(6)=0, only arrays <i>ja</i>, <i>ia</i> are generated for the output storage.</p>
<i>m</i>	<p>INTEGER. Actual row dimension of the matrix A for convert to the BSR format; block row dimension of the matrix A for convert to the CSR format.</p>
<i>mblk</i>	<p>INTEGER. Size of the block in the matrix A.</p>
<i>ldabsr</i>	<p>INTEGER. Leading dimension of the array <i>absr</i> as declared in the calling program. <i>ldabsr</i> must be greater than or equal to <i>mblk</i>*<i>mblk</i>.</p>
<i>acsr</i>	<p>(input/output)</p> <p>REAL for mkl_scsrbsr. DOUBLE PRECISION for mkl_dcsrbsr. COMPLEX for mkl_ccsrbsr.</p>

DOUBLE COMPLEX for `mkl_zcsrbsr`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja (input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia (input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array *acsr*, such that *ia*(*I*) is the index in the array *acsr* of the first non-zero element from the row *I*. The value of the last element *ia*($m + 1$) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

absr (input/output)

REAL for `mkl_scsrbsr`.

DOUBLE PRECISION for `mkl_dcsrbsr`.

COMPLEX for `mkl_ccsrbsr`.

DOUBLE COMPLEX for `mkl_zcsrbsr`.

Array containing elements of non-zero blocks of the matrix *A*. Its length is equal to the number of non-zero blocks in the matrix *A* multiplied by *mblk***mblk*. Refer to *values* array description in [BSR Format](#) for more details.

jab (input/output) INTEGER. Array containing the column indices for each non-zero block of the matrix *A*.

Its length is equal to the number of non-zero blocks of the matrix *A*. Refer to *columns* array description in [BSR Format](#) for more details.

iab (input/output) INTEGER. Array of length $(m + 1)$, containing indices of blocks in the array *absr*, such that *iab*(*i*) is the index in the array *absr* of the first non-zero element from the *i*-th row. The value of the last element *iab*($m + 1$) is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in [BSR Format](#) for more details.

Output Parameters

info INTEGER. Integer info indicator only for converting the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=1, it means that *mblk* is equal to 0.

If *info*=2, it means that *ldabsr* is less than *mblk***mblk* and there is no space for all blocks.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
INTEGER job(8)
```

```
INTEGER m, mblk, ldabsr, info
```

```
INTEGER ja(*), ia(m+1), jab(*), iab(*)
```

```
REAL acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
INTEGER job(8)
```

```
INTEGER m, mblk, ldabsr, info
```

```
INTEGER ja(*), ia(m+1), jab(*), iab(*)
```

```
DOUBLE PRECISION acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
INTEGER job(8)
```

```
INTEGER m, mblk, ldabsr, info
```

```
INTEGER ja(*), ia(m+1), jab(*), iab(*)
```

```
COMPLEX acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
INTEGER job(8)
```

```
INTEGER m, mblk, ldabsr, info
```

```
INTEGER ja(*), ia(m+1), jab(*), iab(*)
```

```
DOUBLE COMPLEX acsr(*), absr(ldabsr,*)
```

mkl_?csrsc

Converts a square sparse matrix in the CSR format to the CSC format and vice versa.

Syntax

```
call mkl_scsrsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

```
call mkl_dcsrsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

```
call mkl_ccsrsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

```
call mkl_zcsrsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

Include Files

- mkl.fi

Description

This routine converts a square sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the compressed sparse column (CSC) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	<p>INTEGER</p> <p>Array, contains the following conversion parameters:</p> <p><i>job</i>(1)</p> <p>If <i>job</i>(1)=0, the matrix in the CSR format is converted to the CSC format; if <i>job</i>(1)=1, the matrix in the CSC format is converted to the CSR format.</p> <p><i>job</i>(2)</p> <p>If <i>job</i>(2)=0, zero-based indexing for the matrix in CSR format is used; if <i>job</i>(2)=1, one-based indexing for the matrix in CSR format is used.</p> <p><i>job</i>(3)</p> <p>If <i>job</i>(3)=0, zero-based indexing for the matrix in the CSC format is used; if <i>job</i>(3)=1, one-based indexing for the matrix in the CSC format is used.</p> <p><i>job</i>(6) - job indicator.</p> <p>For conversion to the CSC format:</p> <p>If <i>job</i>(6)=0, only arrays <i>ja1</i>, <i>ia1</i> are filled in for the output storage. If <i>job</i>(6)≠0, all output arrays <i>acsc</i>, <i>ja1</i>, and <i>ia1</i> are filled in for the output storage.</p> <p>For conversion to the CSR format:</p> <p>If <i>job</i>(6)=0, only arrays <i>ja</i>, <i>ia</i> are filled in for the output storage. If <i>job</i>(6)≠0, all output arrays <i>acsr</i>, <i>ja</i>, and <i>ia</i> are filled in for the output storage.</p>
<i>m</i>	INTEGER. Dimension of the square matrix <i>A</i> .
<i>acsr</i>	<p>(input/output)</p> <p>REAL for mkl_scsrsc.</p> <p>DOUBLE PRECISION for mkl_dcsrsc.</p> <p>COMPLEX for mkl_ccsrsc.</p> <p>DOUBLE COMPLEX for mkl_zcsrsc.</p> <p>Array containing non-zero elements of the square matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>(input/output)INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to the length of the array <i>acsr</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>

<i>ia</i>	(input/output)INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>acsr</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>acsr</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> ($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>acsc</i>	(input/output) REAL for mkl_scsrsc. DOUBLE PRECISION for mkl_dcsrsc. COMPLEX for mkl_ccsrsc. DOUBLE COMPLEX for mkl_zcsrsc. Array containing non-zero elements of the square matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja1</i>	(input/output)INTEGER. Array containing the row indices for each non-zero element of the matrix A. Its length is equal to the length of the array <i>acsc</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia1</i>	(input/output)INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>acsc</i> , such that <i>ia1</i> (<i>i</i>) is the index in the array <i>acsc</i> of the first non-zero element from the column <i>i</i> . The value of the last element <i>ia1</i> ($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.

Output Parameters

<i>info</i>	INTEGER. This parameter is not used now.
-------------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), ja1(*), ia1(m+1)
```

```
REAL       acsr(*), acsc(*)
```

```
SUBROUTINE mkl_dcsrsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), ja1(*), ia1(m+1)
```

```
DOUBLE PRECISION  acsr(*), acsc(*)
```

```
SUBROUTINE mkl_ccsrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

```
INTEGER job(8)
```

```
INTEGER m, info
```

```
INTEGER ja(*), ia(m+1), jal(*), ial(m+1)
```

```
COMPLEX acsr(*), acsc(*)
```

```
SUBROUTINE mkl_zcsrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

```
INTEGER job(8)
```

```
INTEGER m, info
```

```
INTEGER ja(*), ia(m+1), jal(*), ial(m+1)
```

```
DOUBLE COMPLEX acsr(*), acsc(*)
```

mkl_?csrdia

Converts a sparse matrix in the CSR format to the diagonal format and vice versa.

Syntax

```
call mkl_scsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
call mkl_dcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
call mkl_ccsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
call mkl_zcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

Include Files

- mkl.fi

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the diagonal format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

job INTEGER

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the diagonal format;

if $job(1)=1$, the matrix in the diagonal format is converted to the CSR format.

$job(2)$

If $job(2)=0$, zero-based indexing for the matrix in CSR format is used;

if $job(2)=1$, one-based indexing for the matrix in CSR format is used.

$job(3)$

If $job(3)=0$, zero-based indexing for the matrix in the diagonal format is used;

if $job(3)=1$, one-based indexing for the matrix in the diagonal format is used.

$job(6)$ - job indicator.

For conversion to the diagonal format:

If $job(6)=0$, diagonals are not selected internally, and $acsr_rem$, ja_rem , ia_rem are not filled in for the output storage.

If $job(6)=1$, diagonals are not selected internally, and $acsr_rem$, ja_rem , ia_rem are filled in for the output storage.

If $job(6)=10$, diagonals are selected internally, and $acsr_rem$, ja_rem , ia_rem are not filled in for the output storage.

If $job(6)=11$, diagonals are selected internally, and csr_rem , ja_rem , ia_rem are filled in for the output storage.

For conversion to the CSR format:

If $job(6)=0$, each entry in the array $adia$ is checked whether it is zero. Zero entries are not included in the array $acsr$.

If $job(6) \neq 0$, each entry in the array $adia$ is not checked whether it is zero.

m

INTEGER. Dimension of the matrix A .

$acsr$

(input/output)

REAL for `mkl_scsrda`.

DOUBLE PRECISION for `mkl_dcsrda`.

COMPLEX for `mkl_ccsrda`.

DOUBLE COMPLEX for `mkl_zcsrda`.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array $acsr$. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array $acsr$, such that $ia(i)$ is the index in the array $acsr$ of the first non-zero element from the row i . The value of the last

element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

adia

(input/output)

REAL for `mkl_scsrdia`.

DOUBLE PRECISION for `mkl_dcsrdia`.

COMPLEX for `mkl_ccsrdia`.

DOUBLE COMPLEX for `mkl_zcsrdia`.

Array of size ($ndiag \times idiag$) containing diagonals of the matrix *A*.

The key point of the storage is that each element in the array *adia* retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom.

ndiag

INTEGER.

Specifies the leading dimension of the array *adia* as declared in the calling (sub)program, must be at least $\max(1, m)$.

distance

INTEGER.

Array of length *idiag*, containing the distances between the main diagonal and each non-zero diagonal to be extracted. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

idiag

INTEGER.

Number of diagonals to be extracted. For conversion to diagonal format on return this parameter may be modified.

acsr_rem, ja_rem, ia_rem

Remainder of the matrix in the CSR format if it is needed for conversion to the diagonal format.

Output Parameters

info

INTEGER. This parameter is not used now.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info, ndiag, idiag
```

```
INTEGER    ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)
```

```
REAL      acsr(*), adia(*), acsr_rem(*)
```

```
SUBROUTINE mkl_dcsrria(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
INTEGER job(8)
```

```
INTEGER m, info, ndiag, idiag
```

```
INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)
```

```
DOUBLE PRECISION acsr(*), adia(*), acsr_rem(*)
```

```
SUBROUTINE mkl_ccsrria(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
INTEGER job(8)
```

```
INTEGER m, info, ndiag, idiag
```

```
INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)
```

```
COMPLEX acsr(*), adia(*), acsr_rem(*)
```

```
SUBROUTINE mkl_zcsrria(job, m, acsr, ja, ia, adia, ndiag, distance, idiag, acsr_rem, ja_rem, ia_rem, info)
```

```
INTEGER job(8)
```

```
INTEGER m, info, ndiag, idiag
```

```
INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)
```

```
DOUBLE COMPLEX acsr(*), adia(*), acsr_rem(*)
```

mkl_?csrsky

Converts a sparse matrix in CSR format to the skyline format and vice versa.

Syntax

```
call mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
call mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
call mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
call mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

Include Files

- mkl.fi

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the skyline format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

```
job INTEGER
```

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the skyline format;

if *job*(1)=1, the matrix in the skyline format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the skyline format is used;

if *job*(3)=1, one-based indexing for the matrix in the skyline format is used.

job(4)

For conversion to the skyline format:

If *job*(4)=0, the upper part of the matrix *A* in the CSR format is converted.

If *job*(4)=1, the lower part of the matrix *A* in the CSR format is converted.

For conversion to the CSR format:

If *job*(4)=0, the matrix is converted to the upper part of the matrix *A* in the CSR format.

If *job*(4)=1, the matrix is converted to the lower part of the matrix *A* in the CSR format.

job(5)

job(5)=*nzmax* - maximum number of the non-zero elements of the matrix *A* if *job*(1)=0.

job(6) - job indicator.

Only for conversion to the skyline format:

If *job*(6)=0, only arrays *pointers* is filled in for the output storage.

If *job*(6)=1, all output arrays *asky* and *pointers* are filled in for the output storage.

m

INTEGER. Dimension of the matrix *A*.

acsr

(input/output)

REAL for *mkl_scsrsky*.

DOUBLE PRECISION for *mkl_dcsrsky*.

COMPLEX for *mkl_ccsrsky*.

DOUBLE COMPLEX for *mkl_zcsrsky*.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja (input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia (input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array *acsr*, such that $ia(i)$ is the index in the array *acsr* of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

asky (input/output)

REAL for `mkl_scsrsky`.

DOUBLE PRECISION for `mkl_dcsrsky`.

COMPLEX for `mkl_ccsrsky`.

DOUBLE COMPLEX for `mkl_zcsrsky`.

Array, for a lower triangular part of A it contains the set of elements from each row starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets. Refer to *values* array description in [Skyline Storage Format](#) for more details.

pointers (input/output) INTEGER.

Array with dimension $(m+1)$, where m is number of rows for lower triangle (columns for upper triangle), $pointers(i) - pointers(1) + 1$ gives the index of element in the array *asky* that is first non-zero element in row (column) i . The value of $pointers(m + 1)$ is set to $nnz + pointers(1)$, where nnz is the number of elements in the array *asky*. Refer to *pointers* array description in [Skyline Storage Format](#) for more details

Output Parameters

info INTEGER. Integer info indicator only for converting the matrix A from the CSR format.

If $info=0$, the execution is successful.

If $info=1$, the routine is interrupted because there is no space in the array *asky* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), pointers(m+1)
```

```
REAL       acsr(*), asky(*)
```

```
SUBROUTINE mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), pointers(m+1)
```

```
DOUBLE PRECISION acsr(*), asky(*)
```

```
SUBROUTINE mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), pointers(m+1)
```

```
COMPLEX    acsr(*), asky(*)
```

```
SUBROUTINE mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
```

```
INTEGER    job(8)
```

```
INTEGER    m, info
```

```
INTEGER    ja(*), ia(m+1), pointers(m+1)
```

```
DOUBLE COMPLEX acsr(*), asky(*)
```

mkl_?csradd

Computes the sum of two matrices stored in the CSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_scsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_dcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_ccsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_zcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

Include Files

- `mk1.fi`

Description

The `mk1_?csradd` routine performs a matrix-matrix operation defined as

$$C := A + \text{beta} * \text{op}(B)$$

where:

A , B , C are the sparse matrices in the CSR format (3-array variation).

$\text{op}(B)$ is one of $\text{op}(B) = B$, or $\text{op}(B) = B'$, or $\text{op}(A) = \text{conjg}(B')$

beta is a scalar.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices A and B are arranged in the increasing order for each row. If not, use the parameter `sort` (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A + \text{beta} * B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A + \text{beta} * B'$.
<i>request</i>	INTEGER. If <i>request</i> =0, the routine performs addition. The memory for the output arrays <i>ic</i> , <i>jc</i> , <i>c</i> must be allocated beforehand. If <i>request</i> =1, the routine only computes the values of the array <i>ic</i> of length $m + 1$. The memory for the <i>ic</i> array must be allocated beforehand. On exit the value $ic(m+1) - 1$ is the actual number of the elements in the arrays <i>c</i> and <i>jc</i> . If <i>request</i> =2, after the routine is called previously with the parameter <i>request</i> =1 and after the output arrays <i>jc</i> and <i>c</i> are allocated in the calling program with length at least $ic(m+1) - 1$, the routine performs addition.
<i>sort</i>	INTEGER. Specifies the type of reordering. If this parameter is not set (default), the routine does not perform reordering. If <i>sort</i> =1, the routine arranges the column indices <i>ja</i> for each row in the increasing order and reorders the corresponding values of the matrix A in the array <i>a</i> . If <i>sort</i> =2, the routine arranges the column indices <i>jb</i> for each row in the increasing order and reorders the corresponding values of the matrix B in the array <i>b</i> .

If `sort=3`, the routine performs reordering for both input matrices *A* and *B*.

m INTEGER. Number of rows of the matrix *A*.

n INTEGER. Number of columns of the matrix *A*.

a REAL for `mkl_scsradd`.
 DOUBLE PRECISION for `mkl_dcsradd`.
 COMPLEX for `mkl_ccsradd`.
 DOUBLE COMPLEX for `mkl_zcsradd`.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *a*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia INTEGER. Array of length $m + 1$, containing indices of elements in the array *a*, such that $ia(i)$ is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element $ia(m + 1)$ is equal to the number of non-zero elements of the matrix *A* plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

beta REAL for `mkl_scsradd`.
 DOUBLE PRECISION for `mkl_dcsradd`.
 COMPLEX for `mkl_ccsradd`.
 DOUBLE COMPLEX for `mkl_zcsradd`.

Specifies the scalar *beta*.

b REAL for `mkl_scsradd`.
 DOUBLE PRECISION for `mkl_dcsradd`.
 COMPLEX for `mkl_ccsradd`.
 DOUBLE COMPLEX for `mkl_zcsradd`.

Array containing non-zero elements of the matrix *B*. Its length is equal to the number of non-zero elements in the matrix *B*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix *B*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *b*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ib INTEGER. Array of length $m + 1$ when *trans* = 'N' or 'n', or $n + 1$ otherwise.

This array contains indices of elements in the array *b*, such that *ib*(*i*) is the index in the array *b* of the first non-zero element from the row *i*. The value of the last element *ib*($m + 1$) or *ib*($n + 1$) is equal to the number of non-zero elements of the matrix *B* plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

nzmax INTEGER. The length of the arrays *c* and *jc*.

This parameter is used only if *request*=0. The routine stops calculation if the number of elements in the result matrix *C* exceeds the specified value of *nzmax*.

Output Parameters

c REAL for mkl_scsradd.
DOUBLE PRECISION for mkl_dcsradd.
COMPLEX for mkl_ccsradd.
DOUBLE COMPLEX for mkl_zcsradd.

Array containing non-zero elements of the result matrix *C*. Its length is equal to the number of non-zero elements in the matrix *C*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jc INTEGER. Array containing the column indices for each non-zero element of the matrix *C*.

The length of this array is equal to the length of the array *c*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ic INTEGER. Array of length $m + 1$, containing indices of elements in the array *c*, such that *ic*(*i*) is the index in the array *c* of the first non-zero element from the row *i*. The value of the last element *ic*($m + 1$) is equal to the number of non-zero elements of the matrix *C* plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

info INTEGER.

If *info*=0, the execution is successful.

If *info*=*I*>0, the routine stops calculation in the *I*-th row of the matrix *C* because number of elements in *C* exceeds *nzmax*.

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
REAL a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_dcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
DOUBLE PRECISION a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_ccsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
COMPLEX a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_zcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
DOUBLE COMPLEX a(*), b(*), c(*), beta
```

mkl_?csrmultcsr

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.

Syntax

```
call mkl_scsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_dcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_ccsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
call mkl_zcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrmultcsr` routine performs a matrix-matrix operation defined as

$$C := \text{op}(A) * B$$

where:

A, B, C are the sparse matrices in the CSR format (3-array variation);

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

You can use the parameter `sort` to perform or not perform reordering of non-zero entries in input and output sparse matrices. The purpose of reordering is to rearrange non-zero entries in compressed sparse row matrix so that column indices in compressed sparse representation are sorted in the increasing order for each row.

The following table shows correspondence between the value of the parameter `sort` and the type of reordering performed by this routine for each sparse matrix involved:

Value of the parameter <i>sort</i>	Reordering of <i>A</i> (arrays <i>a, ja, ia</i>)	Reordering of <i>B</i> (arrays <i>b, ja, ib</i>)	Reordering of <i>C</i> (arrays <i>c, jc, ic</i>)
1	yes	no	yes
2	no	yes	yes
3	yes	yes	yes
4	yes	no	no
5	no	yes	no
6	yes	yes	no
7	no	no	no
arbitrary value not equal to 1, 2, ..., 7	no	no	yes

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A * B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A' * B$.
<i>request</i>	INTEGER. If <i>request</i> =0, the routine performs multiplication, the memory for the output arrays <i>ic, jc, c</i> must be allocated beforehand.

If $request=1$, the routine computes only values of the array ic of length $m + 1$, the memory for this array must be allocated beforehand. On exit the value $ic(m+1) - 1$ is the actual number of the elements in the arrays c and jc .

If $request=2$, the routine has been called previously with the parameter $request=1$, the output arrays jc and c are allocated in the calling program and they are of the length $ic(m+1) - 1$ at least.

<i>sort</i>	INTEGER. Specifies whether the routine performs reordering of non-zeros entries in input and/or output sparse matrices (see table above).
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>B</i> .
<i>a</i>	<p>REAL for <code>mkl_scsrmultcsr</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmultcsr</code>.</p> <p>COMPLEX for <code>mkl_ccsrmultcsr</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrmultcsr</code>.</p> <p>Array containing non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i>. For each row the column indices must be arranged in the increasing order.</p> <p>The length of this array is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ia</i>	<p>INTEGER. Array of length $m + 1$.</p> <p>This array contains indices of elements in the array <i>a</i>, such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element $ia(m + 1)$ is equal to the number of non-zero elements of the matrix <i>A</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>b</i>	<p>REAL for <code>mkl_scsrmultcsr</code>.</p> <p>DOUBLE PRECISION for <code>mkl_dcsrmultcsr</code>.</p> <p>COMPLEX for <code>mkl_ccsrmultcsr</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zcsrmultcsr</code>.</p> <p>Array containing non-zero elements of the matrix <i>B</i>. Its length is equal to the number of non-zero elements in the matrix <i>B</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>

<i>jb</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix <i>B</i>. For each row the column indices must be arranged in the increasing order.</p> <p>The length of this array is equal to the length of the array <i>b</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ib</i>	<p>INTEGER. Array of length $n + 1$ when <i>trans</i> = 'N' or 'n', or $m + 1$ otherwise.</p> <p>This array contains indices of elements in the array <i>b</i>, such that <i>ib</i>(<i>i</i>) is the index in the array <i>b</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ib</i>($n + 1$) or <i>ib</i>($m + 1$) is equal to the number of non-zero elements of the matrix <i>B</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>nzmax</i>	<p>INTEGER. The length of the arrays <i>c</i> and <i>jc</i>.</p> <p>This parameter is used only if <i>request</i>=0. The routine stops calculation if the number of elements in the result matrix <i>C</i> exceeds the specified value of <i>nzmax</i>.</p>

Output Parameters

<i>c</i>	<p>REAL for mkl_scsrmultcsr. DOUBLE PRECISION for mkl_dcsrmultcsr. COMPLEX for mkl_ccsrmultcsr. DOUBLE COMPLEX for mkl_zcsrmultcsr.</p> <p>Array containing non-zero elements of the result matrix <i>C</i>. Its length is equal to the number of non-zero elements in the matrix <i>C</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>jc</i>	<p>INTEGER. Array containing the column indices for each non-zero element of the matrix <i>C</i>.</p> <p>The length of this array is equal to the length of the array <i>c</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ic</i>	<p>INTEGER. Array of length $m + 1$ when <i>trans</i> = 'N' or 'n', or $n + 1$ otherwise.</p> <p>This array contains indices of elements in the array <i>c</i>, such that <i>ic</i>(<i>i</i>) is the index in the array <i>c</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ic</i>($m + 1$) or <i>ic</i>($n + 1$) is equal to the number of non-zero elements of the matrix <i>C</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i>=<i>I</i>>0, the routine stops calculation in the <i>I</i>-th row of the matrix <i>C</i> because number of elements in <i>C</i> exceeds <i>nzmax</i>.</p>

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER*1  trans
```

```
INTEGER      request, sort, m, n, k, nzmax, info
```

```
INTEGER      ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
REAL         a(*), b(*), c(*)
```

```
SUBROUTINE mkl_dcscrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER*1  trans
```

```
INTEGER      request, sort, m, n, k, nzmax, info
```

```
INTEGER      ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
DOUBLE PRECISION  a(*), b(*), c(*)
```

```
SUBROUTINE mkl_ccscrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER*1  trans
```

```
INTEGER      request, sort, m, n, k, nzmax, info
```

```
INTEGER      ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
COMPLEX      a(*), b(*), c(*)
```

```
SUBROUTINE mkl_zcsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax, info)
```

```
CHARACTER*1  trans
```

```
INTEGER      request, sort, m, n, k, nzmax, info
```

```
INTEGER      ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
DOUBLE COMPLEX  a(*), b(*), c(*)
```

mkl_?csrultd

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

Syntax

```
call mkl_scsrultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
call mkl_dcscrultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
call mkl_ccscrultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
call mkl_zcsrultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

Include Files

- `mkl.fi`

Description

The `mkl_?csrmultd` routine performs a matrix-matrix operation defined as

$$C := \text{op}(A) * B$$

where:

A , B are the sparse matrices in the CSR format (3-array variation), C is dense matrix;

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices A and B are arranged in the increasing order for each row.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A * B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A' * B$.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix A .
<i>k</i>	INTEGER. Number of columns of the matrix B .
<i>a</i>	REAL for <code>mkl_scsrmultd</code> . DOUBLE PRECISION for <code>mkl_dcsrmultd</code> . COMPLEX for <code>mkl_ccsrmultd</code> . DOUBLE COMPLEX for <code>mkl_zcsrmultd</code> . Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix A . For each row the column indices must be arranged in the increasing order. The length of this array is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$ when <i>trans</i> = 'N' or 'n', or $n + 1$ otherwise.

This array contains indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ or $ia(n + 1)$ is equal to the number of non-zero elements of the matrix A plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

b REAL for mkl_scsrmultd.
 DOUBLE PRECISION for mkl_dcscrmultd.
 COMPLEX for mkl_ccscrmultd.
 DOUBLE COMPLEX for mkl_zcscrmultd.

Array containing non-zero elements of the matrix B . Its length is equal to the number of non-zero elements in the matrix B . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix B . For each row the column indices must be arranged in the increasing order.
 The length of this array is equal to the length of the array b . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ib INTEGER. Array of length $m + 1$.
 This array contains indices of elements in the array b , such that $ib(i)$ is the index in the array b of the first non-zero element from the row i . The value of the last element $ib(m + 1)$ is equal to the number of non-zero elements of the matrix B plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

c REAL for mkl_scsrmultd.
 DOUBLE PRECISION for mkl_dcscrmultd.
 COMPLEX for mkl_ccscrmultd.
 DOUBLE COMPLEX for mkl_zcscrmultd.
 Array containing non-zero elements of the result matrix C .

ldc INTEGER. Specifies the leading dimension of the dense matrix C as declared in the calling (sub)program. Must be at least $\max(m, 1)$ when *trans* = 'N' or 'n', or $\max(1, n)$ otherwise.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
CHARACTER*1 trans
```

```
INTEGER m, n, k, ldc
```

```
INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
REAL a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_dcsrmltd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
CHARACTER*1 trans
```

```
INTEGER m, n, k, ldc
```

```
INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
DOUBLE PRECISION a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_ccsrmltd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
CHARACTER*1 trans
```

```
INTEGER m, n, k, ldc
```

```
INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
COMPLEX a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_zcsrmltd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
CHARACTER*1 trans
```

```
INTEGER m, n, k, ldc
```

```
INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
DOUBLE COMPLEX a(*), b(*), c(ldc, *)
```

Inspector-executor Sparse BLAS Routines

The inspector-executor API for Sparse BLAS divides operations into two stages: analysis and execution. During the initial analysis stage, the API inspects the matrix sparsity pattern and applies matrix structure changes. In the execution stage, subsequent routine calls reuse this information in order to improve performance.

The inspector-executor API supports key Sparse BLAS operations for iterative sparse solvers:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with a sparse or dense result
- Solution of triangular systems
- Sparse matrix addition

Naming conventions in Inspector-executor Sparse BLAS Routines

The Inspector-executor Sparse BLAS API routine names use the following convention:

```
mkl_sparse_ [<character>_ ]<operation>[_<format>]
```

The *<character>* field indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The data type is included in the name only if the function accepts dense matrix or scalar floating point parameters.

The *<operation>* field indicates the type of operation:

<code>create</code>	create matrix handle
<code>copy</code>	create a copy of matrix handle
<code>convert</code>	convert matrix between sparse formats
<code>export</code>	export matrix from internal representation to CSR or BSR format
<code>destroy</code>	frees memory allocated for matrix handle
<code>set_<op>_hint</code>	provide information about number of upcoming compute operations and operation type for optimization purposes, where <op> is <code>mv</code> , <code>sv</code> , <code>mm</code> , <code>sm</code> , or <code>memory</code>
<code>optimize</code>	analyze the matrix using hints and store optimization information in matrix handle
<code>mv</code>	compute sparse matrix-vector product
<code>mm</code>	compute sparse matrix by dense matrix product (batch <code>mv</code>)
<code>sppmm/sppmmd</code>	compute sparse matrix by sparse matrix product and store the result as a sparse/dense matrix
<code>trsv</code>	solve a triangular system
<code>trsm</code>	solve a triangular system with multiple right-hand sides
<code>add</code>	compute sum of two sparse matrices

The `<format>` field indicates the sparse matrix storage format:

<code>coo</code>	coordinate format
<code>csr</code>	compressed sparse row format plus variations
<code>csc</code>	compressed sparse column format plus variations
<code>bsr</code>	block sparse row format plus variations

The format is included in the function name only if the function parameters include an explicit sparse matrix in one of the conventional sparse matrix formats.

Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS Routines

Inspector-executor Sparse BLAS routines support four conventional sparse matrix storage formats:

- compressed sparse row format (CSR) plus variations
- compressed sparse column format (CSC) plus variations
- coordinate format (COO)
- block sparse row format (BSR) plus variations

Computational routines operate on a matrix handle that stores a matrix in CSR or BSR formats. Other formats should be converted to CSR or BSR format before calling any computational routines. For more information see [Sparse Matrix Storage Formats](#).

Supported Inspector-executor Sparse BLAS Operations

The Inspector-executor Sparse BLAS API can perform several operations involving sparse matrices. These notations are used in the description of the operations:

- A, G, V are sparse matrices
- B and C are dense matrices
- x and y are dense vectors
- $alpha$ and $beta$ are scalars

$op(A)$ represents a possible transposition of matrix A

$$op(A) = A$$

$$op(A) = A^T - \text{transpose of } A$$

$$op(A) = A^H - \text{conjugate transpose of } A$$

$op(A)^{-1}$ denotes the inverse of $op(A)$.

The Inspector-executor Sparse BLAS routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

$$y := alpha * op(A) * x + beta * y$$

- solving a single triangular system:

$$y := alpha * inv(op(A)) * x$$

- computing a product between a sparse matrix and a dense matrix:

$$C := alpha * op(A) * B + beta * C$$

- computing a product between sparse matrices with a sparse result:

$$V := alpha * op(A) * G$$

- computing a product between sparse matrices with a dense result:

$$C := alpha * op(A) * G$$

- computing a sum of sparse matrices with a sparse result:

$$V := alpha * op(A) + G$$

- solving a sparse triangular system with multiple right-hand sides:

$$C := alpha * inv(op(A)) * B$$

Matrix manipulation routines

The [Matrix Manipulation routines](#) table lists the Matrix Manipulation routines and the data types associated with them.

Matrix Manipulation Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse?_create_csr	s, d, c, z	Creates a handle for a CSR format matrix.
mkl_sparse?_create_csc	s, d, c, z	Creates a handle for a CSC format matrix.
mkl_sparse?_create_coo	s, d, c, z	Creates a handle for a matrix in COO format.
mkl_sparse?_create_bsr	s, d, c, z	Creates a handle for a matrix in BSR format.
mkl_sparse_copy	NA	Creates a copy of a matrix handle.
mkl_sparse_destro y	NA	Frees memory allocated for matrix handle.
mkl_sparse_conve rt_csr	NA	Converts internal matrix representation to CSR format.

Routine or Function Group	Data Types	Description
mkl_sparse_convert_bsr	NA	Converts internal matrix representation to BSR format or changes BSR block size.
mkl_sparse_?_export_csr	s, d, c, z	Exports CSR matrix from internal representation.
mkl_sparse_?_export_bsr	s, d, c, z	Exports BSR matrix from internal representation.
mkl_sparse_?_set_value	s, d, c, z	Changes a single value of matrix in internal representation.

[mkl_sparse_?_create_csr](#)

Creates a handle for a CSR format matrix.

Syntax

```
stat = mkl_sparse_s_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_d_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_c_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_z_create_csr (A, indexing, rows, cols, rows_start, rows_end, col_indx, values)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_create_csr` routine creates a handle for an m -by- k matrix A in CSR format.

Input Parameters

<code>indexing</code>	C_INT. Indicates how input arrays are indexed. SPARSE_INDEX_BASE_Z ERO Zero-based (C-style) indexing: indices start at 0. SPARSE_INDEX_BASE_ONE NE One-based (Fortran-style) indexing: indices start at 1.
<code>rows</code>	C_INT. Number of rows of matrix A .
<code>cols</code>	C_INT. Number of columns of matrix A .
<code>rows_start</code>	C_INT.

Array of length at least m . This array contains row indices, such that $rows_start(i) - rows_start(1)$ is the first index of row i in the arrays *values* and *col_indx*.

Refer to *pointerb* array description in [CSR Format](#) for more details.

rows_end

C_INT.

Array of at least length m . This array contains row indices, such that $rows_end(i) - rows_start(1) - 1$ is the last index of row i in the arrays *values* and *col_indx*.

Refer to *pointerE* array description in [CSR Format](#) for more details.

col_indx

C_INT .

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix A . For zero-based indexing, array containing the column indices for each non-zero element of the matrix A . Its length is at least $rows_end(rows - 1) - rows_start(1)$.

values

C_FLOAT for `mkl_sparse_s_create_csr`

C_DOUBLE for `mkl_sparse_d_create_csr`

C_FLOAT_COMPLEX for `mkl_sparse_c_create_csr`

C_DOUBLE_COMPLEX for `mkl_sparse_z_create_csr`

Array containing non-zero elements of the matrix A . Its length is equal to length of the *col_indx* array.

Refer to *values* array description in [CSR Format](#) for more details.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data for subsequent Inspector-executor Sparse BLAS operations.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.

SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
_FAILED

SPARSE_STATUS_INVAL The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
 SUPPORTED

mkl_sparse_create_csc

Creates a handle for a CSC format matrix.

Syntax

```
stat = mkl_sparse_s_create_csc (A, indexing, rows, cols, rows_start, rows_end,
col_indx, values)
```

```
stat = mkl_sparse_d_create_csc (A, indexing, rows, cols, rows_start, rows_end,
col_indx, values)
```

```
stat = mkl_sparse_c_create_csc (A, indexing, rows, cols, rows_start, rows_end,
col_indx, values)
```

```
stat = mkl_sparse_z_create_csc (A, indexing, rows, cols, rows_start, rows_end,
col_indx, values)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_create_csc` routine creates a handle for an m -by- k matrix A in CSC format.

Input Parameters

<i>indexing</i>	C_INT. Indicates how input arrays are indexed. SPARSE_INDEX_BASE_Z Zero-based (C-style) indexing: indices start at ERO 0. SPARSE_INDEX_BASE_O One-based (Fortran-style) indexing: indices NE start at 1.
<i>rows</i>	C_INT. Number of rows of the matrix A .
<i>cols</i>	C_INT. Number of columns of the matrix A .
<i>rows_start</i>	C_INT. Array of length at least m . This array contains row indices, such that $rows_start(i) - rows_start(1)$ is the first index of row i in the arrays <i>values</i> and <i>col_indx</i> . Refer to <i>pointerb</i> array description in CSC Format for more details.
<i>rows_end</i>	C_INT.

Array of at least length m . This array contains row indices, such that $rows_end(i) - rows_start(1) - 1$ is the last index of row i in the arrays *values* and *col_indx*.

Refer to *pointerE* array description in [CSC Format](#) for more details.

col_indx

C_INT.

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix A . For zero-based indexing, array containing the column indices for each non-zero element of the matrix A . Its length is at least $rows_end(rows - 1) - rows_start(1)$.

values

C_FLOAT for `mkl_sparse_s_create_csc`

C_DOUBLE for `mkl_sparse_d_create_csc`

C_FLOAT_COMPLEX for `mkl_sparse_c_create_csc`

C_DOUBLE_COMPLEX for `mkl_sparse_z_create_csc`

Array containing non-zero elements of the matrix A . Its length is equal to length of the *col_indx* array.

Refer to *values* array description in [CSC Format](#) for more details.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.

SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
_FAILED

SPARSE_STATUS_INVAL The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_S The requested operation is not supported.
UPPORTED

`mkl_sparse_?_create_coo`

Creates a handle for a matrix in COO format.

Syntax

```
stat = mkl_sparse_s_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
```

```
stat = mkl_sparse_d_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
```

```
stat = mkl_sparse_c_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
```

```
stat = mkl_sparse_z_create_coo (A, indexing, rows, cols, nnz, row_indx, col_indx, values)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_create_coo` routine creates a handle for an m -by- k matrix A in COO format.

Input Parameters

<i>indexing</i>	C_INT. Indicates how input arrays are indexed. SPARSE_INDEX_BASE_Z Zero-based (C-style) indexing: indices start at 0. ERO SPARSE_INDEX_BASE_O One-based (Fortran-style) indexing: indices start at 1. NE
<i>rows</i>	C_INT. Number of rows of matrix A .
<i>cols</i>	C_INT. Number of columns of matrix A .
<i>nnz</i>	C_INT. Specifies the number of non-zero elements of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>row_indx</i>	C_INT. Array of length <i>nnz</i> , containing the row indices for each non-zero element of matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>col_indx</i>	C_INT. Array of length <i>nnz</i> , containing the column indices for each non-zero element of matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>values</i>	C_FLOAT for <code>mkl_sparse_s_create_coo</code>

C_DOUBLE for `mkl_sparse_d_create_coo`
 C_FLOAT_COMPLEX for `mkl_sparse_c_create_coo`
 C_DOUBLE_COMPLEX for `mkl_sparse_z_create_coo`

Array of length *nnz*, containing the non-zero elements of matrix *A* in arbitrary order.

Refer to *values* array description in [Coordinate Format](#) for more details.

Output Parameters

A SPARSE_MATRIX_T.
 Handle containing internal data.

stat INTEGER.
 Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.
 SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
 INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
 _FAILED

SPARSE_STATUS_INVALID The input parameters contain an invalid value.
 ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
 TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
 NAL_ERROR

SPARSE_STATUS_NOT_S The requested operation is not supported.
 UPPORTED

`mkl_sparse_?_create_bsr`

Creates a handle for a matrix in BSR format.

Syntax

```
stat = mkl_sparse_s_create_bsr (A, indexing, block_layout, rows, cols, block_size,
rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_d_create_bsr (A, indexing, block_layout, rows, cols, block_size,
rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_c_create_bsr (A, indexing, block_layout, rows, cols, block_size,
rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_z_create_bsr (A, indexing, block_layout, rows, cols, block_size,
rows_start, rows_end, col_indx, values)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_create_bsr` routine creates a handle for an m -by- k matrix A in BSR format.

Input Parameters

<code>indexing</code>	C_INT. Indicates how input arrays are indexed. SPARSE_INDEX_BASE_Z ERO Zero-based (C-style) indexing: indices start at 0. SPARSE_INDEX_BASE_ONE NE One-based (Fortran-style) indexing: indices start at 1.
<code>block_layout</code>	C_INT. Specifies layout of blocks: SPARSE_LAYOUT_ROW_MAJOR AJOR Storage of elements of blocks uses row major layout. SPARSE_LAYOUT_COLUMN_MAJOR N_MAJOR Storage of elements of blocks uses column major layout.
<code>rows</code>	C_INT. Number of block rows of matrix A .
<code>cols</code>	C_INT. Number of block columns of matrix A .
<code>block_size</code>	C_INT. Size of blocks in matrix A .
<code>rows_start</code>	C_INT. Array of length m . This array contains row indices, such that $rows_start(i) - rows_start(1)$ is the first index of block row i in the arrays <code>values</code> and <code>col_indx</code> . Refer to <code>pointerb</code> array description in CSR Format for more details.
<code>rows_end</code>	C_INT. Array of length m . This array contains row indices, such that $rows_end(i) - rows_start(1) - 1$ is the last index of block row i in the arrays <code>values</code> and <code>col_indx</code> . Refer to <code>pointerE</code> array description in CSR Format for more details.
<code>col_indx</code>	C_INT.

For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix *A*. For zero-based indexing, array containing the column indices for each non-zero block of the matrix *A*. Its length is `rows_end(rows - 1) - rows_start(1)`.

values

C_FLOAT for `mkl_sparse_s_create_bsr`

C_DOUBLE for `mkl_sparse_d_create_bsr`

C_FLOAT_COMPLEX for `mkl_sparse_c_create_bsr`

C_DOUBLE_COMPLEX for `mkl_sparse_z_create_bsr`

Array containing non-zero elements of the matrix *A*. Its length is equal to length of the `col_indx` array multiplied by `block_size*block_size`.

Refer to the *values* array description in [BSR Format](#) for more details.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.
SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
_FAILED

SPARSE_STATUS_INVAL The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_S The requested operation is not supported.
UPPORTED

mkl_sparse_copy

Creates a copy of a matrix handle.

Syntax

```
stat = mkl_sparse_copy (source, descr, dest)
```

Include Files

- `mkl_splblas.f90`

Description

The `mkl_sparse_copy` routine creates a copy of a matrix handle.

Input Parameters

<code>source</code>	<code>SPARSE_MATRIX_T</code> . Specifies handle containing internal data.																						
<code>descr</code>	<code>MATRIX_DESCR</code> . Descriptor specifying sparse matrix properties. <i>type</i> - Specifies the type of a sparse matrix: <table> <tr> <td><code>SPARSE_MATRIX_TYPE_GENERAL</code></td> <td>The matrix is processed as is.</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_SYMMETRIC</code></td> <td>The matrix is symmetric (only the requested triangle is processed).</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_HERMITIAN</code></td> <td>The matrix is Hermitian (only the requested triangle is processed).</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_TRIANGULAR</code></td> <td>The matrix is triangular (only the requested triangle is processed).</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_DIAGONAL</code></td> <td>The matrix is diagonal (only diagonal elements are processed).</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code></td> <td>The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)</td> </tr> <tr> <td><code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code></td> <td>The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)</td> </tr> </table> <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: <table> <tr> <td><code>SPARSE_FILL_MODE_LOWER</code></td> <td>The lower triangular matrix part is processed.</td> </tr> <tr> <td><code>SPARSE_FILL_MODE_UPPER</code></td> <td>The upper triangular matrix part is processed.</td> </tr> </table> <i>diag</i> - Specifies diagonal type for non-general matrices: <table> <tr> <td><code>SPARSE_DIAG_NON_UNIT</code></td> <td>Diagonal elements might not be equal to one.</td> </tr> <tr> <td><code>SPARSE_DIAG_UNIT</code></td> <td>Diagonal elements are equal to one.</td> </tr> </table>	<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).	<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)	<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)	<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.	<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.	<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.	<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.																						
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).																						
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).																						
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).																						
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).																						
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)																						
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)																						
<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.																						
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.																						
<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.																						
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.																						

Output Parameters

<code>dest</code>	<code>SPARSE_MATRIX_T</code> for <code>mkl_sparse_copy</code> Handle containing internal data.
-------------------	---

`stat` INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.
SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECUTION_FAILED Execution failed.
TION_FAILED

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
UPPORTED

mkl_sparse_destroy

Frees memory allocated for matrix handle.

Syntax

`stat = mkl_sparse_destroy (A)`

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_destroy` routine frees memory allocated for matrix handle.

NOTE

You must free memory allocated for matrices after completing use of them. The `mkl_sparse_destroy` provides a utility to do so.

Input Parameters

`A` SPARSE_MATRIX_T.
Handle containing internal data.

Output Parameters

`stat` INTEGER.
Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_convert_csr

Converts internal matrix representation to CSR format.

Syntax

```
stat = mkl_sparse_convert_csr (source, operation, dest)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_convert_csr` routine converts internal matrix representation to CSR format.

Input Parameters

<i>source</i>	SPARSE_MATRIX_T. Handle containing internal data.
<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix.
	SPARSE_OPERATION_NO_TRANSPOSE Non-transpose, $op(A) = A$.
	SPARSE_OPERATION_TRANSPOSE Transpose, $op(A) = A^T$.
	SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $op(A) = A^H$.

Output Parameters

<i>dest</i>	SPARSE_MATRIX_T.
-------------	------------------

Handle containing internal data.

stat INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.
SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECUTION_FAILED Execution failed.
TION_FAILED

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
UPPORTED

mkl_sparse_convert_bsr

Converts internal matrix representation to BSR format or changes BSR block size.

Syntax

```
stat = mkl_sparse_convert_bsr (source, block_size, block_layout, operation, dest)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_convert_bsr` routine converts internal matrix representation to BSR format or changes BSR block size.

Input Parameters

source SPARSE_MATRIX_T.
Handle containing internal data.

block_size C_INT.
Size of the block in the output structure.

block_layout C_INT.
Specifies layout of blocks:
SPARSE_LAYOUT_ROW_MAJOR Storage of elements of blocks uses row major layout.
AJOR

SPARSE_LAYOUT_COLUMN_STORAGE Storage of elements of blocks uses column major layout.
 N_MAJOR

operation

C_INT.

Specifies operation `op()` on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, $op(A) = A$.
 N_TRANSPOSE

SPARSE_OPERATION_TRANSPOSE Transpose, $op(A) = A^T$.
 ANSPOSE

SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $op(A) = A^H$.
 NJUGATE_TRANSPOSE

Output Parameters

dest

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.
 SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.
 _FAILED

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
 ID_VALUE

SPARSE_STATUS_EXECUTION_FAILED Execution failed.
 TION_FAILED

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
 NAL_ERROR

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
 UPPORTED

mkl_sparse?_export_csr

Exports CSR matrix from internal representation.

Syntax

`stat = mkl_sparse_s_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)`

`stat = mkl_sparse_d_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)`

`stat = mkl_sparse_c_export_csr (source, indexing, rows, cols, rows_start, rows_end, col_indx, values)`

Refer to *pointerE* array description in [CSR Format](#) for more details.

col_indx

C_INT.

For one-based indexing, pointer to array containing the column indices plus one for each non-zero element of the matrix *source*. For zero-based indexing, pointer to array containing the column indices for each non-zero element of the matrix *source*. Its length is $rows_end(rows - 1) - rows_start(1)$.

values

C_FLOAT for `mkl_sparse_s_export_csr`

C_DOUBLE for `mkl_sparse_d_export_csr`

C_FLOAT_COMPLEX for `mkl_sparse_c_export_csr`

C_DOUBLE_COMPLEX for `mkl_sparse_z_export_csr`

Pointer to array containing non-zero elements of the matrix A. Its length is equal to length of the *col_indx* array.

Refer to *values* array description in [CSR Format](#) for more details.

Output Parameters

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.

SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
_FAILED

SPARSE_STATUS_INVAL The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_S The requested operation is not supported.
UPPORTED

mkl_sparse_?_export_bsr

Exports BSR matrix from internal representation.

Syntax

```
stat = mkl_sparse_s_export_bsr (source, indexing, block_layout, rows, cols,  
block_size, rows_start, rows_end, col_indx, values)
```

```
stat = mkl_sparse_d_export_bsr (source, indexing, block_layout, rows, cols,  
block_size, rows_start, rows_end, col_indx, values)
```


<i>block_size</i>	<p>C_INT.</p> <p>Size of the block in matrix <i>source</i>.</p>
<i>rows_start</i>	<p>C_INT.</p> <p>Pointer to array of length <i>m</i>. This array contains row indices, such that $rows_start(i) - rows_start(1)$ is the first index of block row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSR Format for more details.</p>
<i>rows_end</i>	<p>C_INT.</p> <p>Pointer to array of length <i>m</i>. This array contains row indices, such that $rows_end(i) - rows_start(1) - 1$ is the last index of block row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>col_indx</i>	<p>C_INT.</p> <p>For one-based indexing, pointer to array containing the column indices plus one for each non-zero blocks of the matrix <i>source</i>. For zero-based indexing, pointer to array containing the column indices for each non-zero blocks of the matrix <i>source</i>. Its length is $rows_end(m - 1) - rows_start(1)$.</p>
<i>values</i>	<p>C_FLOAT for <code>mkl_sparse_s_export_bsr</code> C_DOUBLE for <code>mkl_sparse_d_export_bsr</code> C_FLOAT_COMPLEX for <code>mkl_sparse_c_export_bsr</code> C_DOUBLE_COMPLEX for <code>mkl_sparse_z_export_bsr</code></p> <p>Pointer to array containing non-zero elements of matrix <i>source</i>. Its length is equal to length of the <i>col_indx</i> array multiplied by $block_size * block_size$.</p> <p>Refer to the <i>values</i> array description in BSR Format for more details.</p>

Output Parameters

<i>stat</i>	<p>INTEGER.</p> <p>Value indicating whether the operation was successful or not, and why:</p> <p>SPARSE_STATUS_SUCCE The operation was successful. SS</p> <p>SPARSE_STATUS_NOT_I The routine encountered an empty handle or INITIALIZED matrix array.</p> <p>SPARSE_STATUS_ALLOC Internal memory allocation failed. _FAILED</p> <p>SPARSE_STATUS_INVALID The input parameters contain an invalid value. ID_VALUE</p> <p>SPARSE_STATUS_EXECU Execution failed. TION_FAILED</p>
-------------	--

SPARSE_STATUS_INTER_NAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_?_set_value

Changes a single value of matrix in internal representation.

Syntax

```
stat = mkl_sparse_s_set_value (A , row, col, value);
stat = mkl_sparse_d_set_value (A, row, col, value );
stat = mkl_sparse_c_set_value (A , row, col, value);
stat = mkl_sparse_z_set_value (A , row, col, value);
```

Include Files

- mkl_spblas.f90

Description

Use the `mkl_sparse_?_set_value` routine to change a single value of a matrix in internal Inspector-executor Sparse BLAS format.

Input Parameters

<i>A</i>	SPARSE_MATRIX_T. Specifies handle containing internal data.
<i>row</i>	C_INT . Indicates row of matrix in which to set value.
<i>col</i>	C_INT . Indicates column of matrix in which to set value.
<i>value</i>	C_FLOAT for mkl_sparse_s_create_csr C_DOUBLE for mkl_sparse_d_create_csr C_FLOAT_COMPLEX for mkl_sparse_c_create_csr C_DOUBLE_COMPLEX for mkl_sparse_z_create_csr Indicates value

Output Parameters

<i>A</i>	Handle containing modified internal data.
<i>stat</i>	INTEGER. Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.
 SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_INVALID_ID_VALUE The input parameters contain an invalid value.
 ID_VALUE

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
 INTERNAL_ERROR

Inspector-executor Sparse BLAS Analysis Routines

Analysis Routines and Their Data Types

Routine or Function Group	Description
mkl_sparse_set_mv_hint	Provides estimate of number and type of upcoming matrix-vector operations.
mkl_sparse_set_sv_hint	Provides estimate of number and type of upcoming triangular system solver operations.
mkl_sparse_set_mm_hint	Provides estimate of number and type of upcoming matrix-matrix multiplication operations.
mkl_sparse_set_sm_hint	Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.
mkl_sparse_set_memory_hint	Provides memory requirements for performance optimization purposes.
mkl_sparse_optimize	Analyzes matrix structure and performs optimizations using the hints provided in the handle.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

[mkl_sparse_set_mv_hint](#)

Provides estimate of number and type of upcoming matrix-vector operations.

Syntax

```
stat = mkl_sparse_set_mv_hint (A, operation, descr, expected_calls)
```

Include Files

- `mkl_spblas.f90`

Description

Use the `mkl_sparse_set_mv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NO Non-transpose, $op(A) = A$. N_TRANSPOSE SPARSE_OPERATION_TR Transpose, $op(A) = A^T$. ANSPOSE SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$. NJUGATE_TRANSPOSE
<i>descr</i>	MATRIX_DESCR. Descriptor specifying sparse matrix properties. <i>type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_ The matrix is processed as is. GENERAL SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested SYMMETRIC triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested HERMITIAN triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is triangular (only the requested TRIANGULAR triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is diagonal (only diagonal elements DIAGONAL are processed). SPARSE_MATRIX_TYPE_ The matrix is block-triangular (only requested BLOCK_TRIANGULAR triangle is processed). (Applies to BSR format only.)

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed. (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

expected_calls

C_INT.

Number of expected calls to execution routine.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_set_sv_hint

Provides estimate of number and type of upcoming triangular system solver operations.

Syntax

```
stat = mkl_sparse_set_sv_hint (A, operation, descr, expected_calls)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_sv_hint` routine provides an estimate of the number of upcoming triangular system solver operations and type of these operations for performance optimization.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

operation

C_INT.

Specifies operation `op()` on input matrix.

SPARSE_OPERATION_NO Non-transpose, $op(A) = A$.
N_TRANSPOSE

SPARSE_OPERATION_TR Transpose, $op(A) = A^T$.
ANSPOSE

SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$.
NJUGATE_TRANSPOSE

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_ The matrix is processed as is.
GENERAL

SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested
SYMMETRIC triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested
HERMITIAN triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is triangular (only the requested
TRIANGULAR triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is diagonal (only diagonal elements
DIAGONAL are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

expected_calls

C_INT.

Number of expected calls to execution routine.

Output Parameters

A

SPARSE_MATRIX_T.

Handle containing internal data.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_set_mm_hint

Provides estimate of number and type of upcoming matrix-matrix multiplication operations.

Syntax

```
stat = mkl_sparse_set_mm_hint (A, operation, descr, layout, dense_matrix_size,
expected_calls)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_set_mm_hint` routine provides an estimate of the number of upcoming matrix-matrix multiplication operations and type of these operations for performance optimization purposes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NO Non-transpose, $op(A) = A$. N_TRANSPOSE SPARSE_OPERATION_TR Transpose, $op(A) = A^T$. ANSPOSE SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$. NJUGATE_TRANSPOSE
<i>descr</i>	MATRIX_DESCR. Descriptor specifying sparse matrix properties. <i>type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_ The matrix is processed as is. GENERAL SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested SYMMETRIC triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested HERMITIAN triangle is processed).

<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.

layout

`C_INT.`

Specifies layout of elements:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.

dense_matrix_size

`C_INT.`

Number of columns in dense matrix.

expected_calls

`C_INT.`

Number of expected calls to execution routine.

Output Parameters

A

`SPARSE_MATRIX_T.`

Handle containing internal data.

stat

`INTEGER.`

Value indicating whether the operation was successful or not, and why:

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SS</code>	

SPARSE_OPERATION_NO Non-transpose, $op(A) = A$.
N_TRANSPOSE

SPARSE_OPERATION_TR Transpose, $op(A) = A^T$.
ANSPOSE

SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$.
NJUGATE_TRANSPOSE

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_ The matrix is processed as is.
GENERAL

SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested
SYMMETRIC triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested
HERMITIAN triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is triangular (only the requested
TRIANGULAR triangle is processed).

SPARSE_MATRIX_TYPE_ The matrix is diagonal (only diagonal elements
DIAGONAL are processed).

SPARSE_MATRIX_TYPE_ The matrix is block-triangular (only requested
BLOCK_TRIANGULAR triangle is processed). (Applies to BSR format
only.)

SPARSE_MATRIX_TYPE_ The matrix is block-diagonal (only diagonal
BLOCK_DIAGONAL blocks are processed). (Applies to BSR format
only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian,
triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LO The lower triangular matrix part is processed.
WER

SPARSE_FILL_MODE_UP The upper triangular matrix part is processed.
PER

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNI Diagonal elements might not be equal to one.
T

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

layout

C_INT.

Specifies layout of elements:

SPARSE_LAYOUT_COLUM Storage of elements uses column major layout.
N_MAJOR

	SPARSE_LAYOUT_ROW_M	Storage of elements uses row major layout.
	AJOR	
<i>dense_matrix_size</i>	C_INT.	
		Number of right-hand-side.
<i>expected_calls</i>	C_INT.	
		Number of expected calls to execution routine.

Output Parameters

<i>A</i>	SPARSE_MATRIX_T.	
		Handle containing internal data.
<i>stat</i>	INTEGER.	
		Value indicating whether the operation was successful or not, and why:
	SPARSE_STATUS_SUCCE	The operation was successful.
	SS	
	SPARSE_STATUS_NOT_I	The routine encountered an empty handle or
	INITIALIZED	matrix array.
	SPARSE_STATUS_ALLOC	Internal memory allocation failed.
	_FAILED	
	SPARSE_STATUS_INVALID	The input parameters contain an invalid value.
	ID_VALUE	
	SPARSE_STATUS_EXECU	Execution failed.
	TION_FAILED	
	SPARSE_STATUS_INTER	An error in algorithm implementation occurred.
	NAL_ERROR	
	SPARSE_STATUS_NOT_S	The requested operation is not supported.
	UPPORTED	

mkl_sparse_set_memory_hint

Provides memory requirements for performance optimization purposes.

Syntax

```
stat = mkl_sparse_set_memory_hint (A, policy)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_set_memory_hint` routine allocates additional memory for further performance optimization purposes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>policy</i>	C_INT.	
		Specify memory utilization policy for optimization routine using these types:
	SPARSE_MEMORY_NONE	Routine can allocate memory only for auxiliary structures (such as for workload balancing); the amount of memory is proportional to vector size.
	SPARSE_MEMORY_AGGRESSIVE	Default. Routine can allocate memory up to the size of matrix A for converting into the appropriate sparse format.

Output Parameters

<i>A</i>	SPARSE_MATRIX_T.	
		Handle containing internal data.
<i>stat</i>	INTEGER.	
		Value indicating whether the operation was successful or not, and why:
	SPARSE_STATUS_SUCCESS	The operation was successful.
	SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
	SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
	SPARSE_STATUS_INVALID_ID_VALUE	The input parameters contain an invalid value.
	SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
	SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
 SUPPORTED

mkl_sparse_optimize

Analyzes matrix structure and performs optimizations using the hints provided in the handle.

Syntax

```
stat = mkl_sparse_optimize (A)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_optimize` routine analyzes matrix structure and performs optimizations using the hints provided in the handle. Generally, specifying a higher number of expected operations allows for more aggressive and time consuming optimizations.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

`A` SPARSE_MATRIX_T.
 Handle containing internal data.

Output Parameters

`stat` INTEGER.
 Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.
 SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
 ID_VALUE

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

Inspector-executor Sparse BLAS Execution Routines

Execution Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse?_mv	s, d, c, z	Computes a sparse matrix-vector product.
mkl_sparse?_trsv	s, d, c, z	Solves a system of linear equations for a square sparse matrix.
mkl_sparse?_mm	s, d, c, z	Computes the product of a sparse matrix and a dense matrix.
mkl_sparse?_trsm	s, d, c, z	Solves a system of linear equations with multiple right hand sides for a square sparse matrix.
mkl_sparse?_add	s, d, c, z	Computes sum of two sparse matrices.
mkl_sparse_spmmm	s, d, c, z	Computes the product of two sparse matrices and stores the result as a sparse matrix.
mkl_sparse?_spmmd	s, d, c, z	Computes the product of two sparse matrices and stores the result as a dense matrix.

[mkl_sparse?_mv](#)

Computes a sparse matrix-vector product.

Syntax

```
stat = mkl_sparse_s_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_d_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_c_mv (operation, alpha, A, descr, x, beta, y)
stat = mkl_sparse_z_mv (operation, alpha, A, descr, x, beta, y)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse?_mv` routine computes a sparse matrix-vector product defined as

$$y := \alpha * \text{op}(A) * x + \beta * y$$

where:

alpha and *beta* are scalars, *x* and *y* are vectors, and *A* is a matrix handle of a matrix with *m* rows and *k* columns.

Input Parameters

<i>operation</i>	C_INT. Specifies operation $op()$ on input matrix. SPARSE_OPERATION_NO Non-transpose, $op(A) = A$. N_TRANSPOSE SPARSE_OPERATION_TR Transpose, $op(A) = A^T$. ANSPOSE SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$. NJUGATE_TRANSPOSE
<i>alpha</i>	C_FLOAT for mkl_sparse_s_mv C_DOUBLE for mkl_sparse_d_mv C_FLOAT_COMPLEX for mkl_sparse_c_mv C_DOUBLE_COMPLEX for mkl_sparse_z_mv Specifies the scalar <i>alpha</i> .
<i>A</i>	SPARSE_MATRIX_T. Handle containing sparse matrix in internal data structure.
<i>descr</i>	MATRIX_DESCR. Descriptor specifying sparse matrix properties. <i>type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_ The matrix is processed as is. GENERAL SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested SYMMETRIC triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested HERMITIAN triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is triangular (only the requested TRIANGULAR triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is diagonal (only diagonal elements DIAGONAL are processed). SPARSE_MATRIX_TYPE_ The matrix is block-triangular (only requested BLOCK_TRIANGULAR triangle is processed). (Applies to BSR format only.) SPARSE_MATRIX_TYPE_ The matrix is block-diagonal (only diagonal BLOCK_DIAGONAL blocks are processed). (Applies to BSR format only.) <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LO The lower triangular matrix part is processed.
 WER

SPARSE_FILL_MODE_UP The upper triangular matrix part is processed.
 PER

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNI Diagonal elements might not be equal to one.
 T

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

x

C_FLOAT for mkl_sparse_s_mv

C_DOUBLE for mkl_sparse_d_mv

C_FLOAT_COMPLEX for mkl_sparse_c_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_mv

Array with size equal to the number of columns of *A* if *operation* = SPARSE_OPERATION_NON_TRANSPOSE and at least rows of *A* otherwise. On entry, the array *x* must contain the vector *x*.

beta

C_FLOAT for mkl_sparse_s_mv

C_DOUBLE for mkl_sparse_d_mv

C_FLOAT_COMPLEX for mkl_sparse_c_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_mv

Specifies the scalar *beta*.

y

C_FLOAT for mkl_sparse_s_mv

C_DOUBLE for mkl_sparse_d_mv

C_FLOAT_COMPLEX for mkl_sparse_c_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_mv

Array with size at least *m* if *operation*=SPARSE_OPERATION_NON_TRANSPOSE and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

C_FLOAT for mkl_sparse_s_mv

C_DOUBLE for mkl_sparse_d_mv

C_FLOAT_COMPLEX for mkl_sparse_c_mv

C_DOUBLE_COMPLEX for mkl_sparse_z_mv

Overwritten by the updated vector *y*.

stat

INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse?_trsv

Solves a system of linear equations for a triangular sparse matrix.

Syntax

```
stat = mkl_sparse_s_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_d_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_c_trsv (operation, alpha, A, descr, x, y)
stat = mkl_sparse_z_trsv (operation, alpha, A, descr, x, y)
```

Include Files

- mkl_splblas.f90

Description

The `mkl_sparse?_trsv` routine solves a system of linear equations for a matrix:

$$\text{op}(A) * y = \text{alpha} * x$$

where A is a triangular sparse matrix, alpha is a scalar, and x and y are vectors.

Input Parameters

operation

C_INT.

Specifies operation `op()` on input matrix.

SPARSE_OPERATION_NO_TRANSPOSE Non-transpose, $\text{op}(A) = A$.

SPARSE_OPERATION_TRANSPOSE Transpose, $\text{op}(A) = A^T$.

SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $\text{op}(A) = A^H$.

alpha C_FLOAT for mkl_sparse_s_trsv
 C_DOUBLE for mkl_sparse_d_trsv
 C_FLOAT_COMPLEX for mkl_sparse_c_trsv
 C_DOUBLE_COMPLEX for mkl_sparse_z_trsv
Specifies the scalar *alpha*.

A SPARSE_MATRIX_T.
Handle containing sparse matrix in internal data structure.

descr MATRIX_DESCR.
Descriptor specifying sparse matrix properties.
type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_ GENERAL The matrix is processed as is.

SPARSE_MATRIX_TYPE_ SYMMETRIC The matrix is symmetric (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_ HERMITIAN The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_ TRIANGULAR The matrix is triangular (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_ DIAGONAL The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_ BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)

SPARSE_MATRIX_TYPE_ BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LO WER The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UP PER The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

x C_FLOAT for mkl_sparse_s_trsv
 C_DOUBLE for mkl_sparse_d_trsv

C_FLOAT_COMPLEX for mkl_sparse_c_trsv

C_DOUBLE_COMPLEX for mkl_sparse_z_trsv

Array of size at least m , where m is the number of rows of matrix A . On entry, the array x must contain the vector x .

Output Parameters

y C_FLOAT for mkl_sparse_s_trsv
 C_DOUBLE for mkl_sparse_d_trsv
 C_FLOAT_COMPLEX for mkl_sparse_c_trsv
 C_DOUBLE_COMPLEX for mkl_sparse_z_trsv

Array of size at least m containing the solution to the system of linear equations.

$stat$ INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCE The operation was successful.
 SS

SPARSE_STATUS_NOT_I The routine encountered an empty handle or
 INITIALIZED matrix array.

SPARSE_STATUS_ALLOC Internal memory allocation failed.
 _FAILED

SPARSE_STATUS_INVAL The input parameters contain an invalid value.
 ID_VALUE

SPARSE_STATUS_EXECU Execution failed.
 TION_FAILED

SPARSE_STATUS_INTER An error in algorithm implementation occurred.
 NAL_ERROR

SPARSE_STATUS_NOT_S The requested operation is not supported.
 UPPORTED

mkl_sparse?_mm

Computes the product of a sparse matrix and a dense matrix.

Syntax

$stat = \text{mkl_sparse_s_mm}(\text{operation}, \text{alpha}, A, \text{descr}, \text{layout}, x, \text{columns}, \text{ldx}, \text{beta}, y, \text{ldy})$

$stat = \text{mkl_sparse_d_mm}(\text{operation}, \text{alpha}, A, \text{descr}, \text{layout}, x, \text{columns}, \text{ldx}, \text{beta}, y, \text{ldy})$

$stat = \text{mkl_sparse_c_mm}(\text{operation}, \text{alpha}, A, \text{descr}, \text{layout}, x, \text{columns}, \text{ldx}, \text{beta}, y, \text{ldy})$

$stat = \text{mkl_sparse_z_mm}(\text{operation}, \text{alpha}, A, \text{descr}, \text{layout}, x, \text{columns}, \text{ldx}, \text{beta}, y, \text{ldy})$

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_mm` routine performs a matrix-matrix operation:

$$y := \alpha * \text{op}(A) * x + \beta * y$$

where *alpha* and *beta* are scalars, *A* and *y* are the sparse matrices, and *x* is a dense matrix.

The `mkl_sparse_?_mm` and `mkl_sparse_?_trsm` routines support these configurations:

	Column-major dense matrix: <i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJ OR	Row-major dense matrix: <i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
0-based sparse matrix: SPARSE_INDEX_BASE_ZERO	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: SPARSE_INDEX_BASE_ONE	All formats	CSR BSR: general non-transposed matrix multiplication only

Input Parameters

operation

C_INT.

Specifies operation `op()` on input matrix.

SPARSE_OPERATION_NO Non-transpose, $\text{op}(A) = A$.
N_TRANSPOSE

SPARSE_OPERATION_TR Transpose, $\text{op}(A) = A^T$.
ANSPOSE

SPARSE_OPERATION_CO Conjugate transpose, $\text{op}(A) = A^H$.
NJUGATE_TRANSPOSE

alpha

C_FLOAT for `mkl_sparse_s_mm`

C_DOUBLE for `mkl_sparse_d_mm`

C_FLOAT_COMPLEX for `mkl_sparse_c_mm`

C_DOUBLE_COMPLEX for `mkl_sparse_z_mm`

Specifies the scalar *alpha*.

A

SPARSE_MATRIX_T.

Handle containing sparse matrix in internal data structure.

descr

MATRIX_DESCR.

Descriptor specifying sparse matrix properties.

type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_ GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_ SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_ HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_ TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_ DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_ BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)
SPARSE_MATRIX_TYPE_ BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LO WER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UP PER	The upper triangular matrix part is processed.

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNI T	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

layout

C_INT.

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN N_MAJOR	Storage of elements uses column major layout.
SPARSE_LAYOUT_ROW_M AJOR	Storage of elements uses row major layout.

x

C_FLOAT for mkl_sparse_s_mm
 C_DOUBLE for mkl_sparse_d_mm
 C_FLOAT_COMPLEX for mkl_sparse_c_mm
 C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Array of size at least *rows*cols*.

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	If $op(A) = A$, number of columns in A If $op(A) = A^T$, number of rows in A
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

columns

C_INT.

Number of columns of matrix *y*.

ldx

C_INT.

Specifies the leading dimension of matrix *x*.

beta

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Specifies the scalar *beta*

y

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Array of size at least *rows*cols*, where

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>y</i>)	<i>ldy</i>	If $op(A) = A$, number of columns in A If $op(A) = A^T$, number of rows in A
<i>cols</i> (number of columns in <i>y</i>)	<i>columns</i>	<i>ldy</i>

Output Parameters

y

C_FLOAT for mkl_sparse_s_mm

C_DOUBLE for mkl_sparse_d_mm

C_FLOAT_COMPLEX for mkl_sparse_c_mm

C_DOUBLE_COMPLEX for mkl_sparse_z_mm

Overwritten by the updated matrix y .

stat INTEGER.

Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS The operation was successful.
SS

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
ID_VALUE

SPARSE_STATUS_EXECUTION_FAILED Execution failed.
TION_FAILED

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
NAL_ERROR

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
UPPORTED

mkl_sparse_?_trsm

Solves a system of linear equations with multiple right hand sides for a triangular sparse matrix.

Syntax

```
stat = mkl_sparse_s_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_d_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_c_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
stat = mkl_sparse_z_trsm (operation, alpha, A, descr, layout, x, columns, ldx, y, ldy)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_?_trsm` routine solves a system of linear equations with multiple right hand sides for a triangular sparse matrix:

$$y := \alpha \cdot \text{inv}(\text{op}(A)) \cdot x$$

where:

α is a scalar, x and y are dense matrices, and A is a sparse matrix.

The `mkl_sparse_?_mm` and `mkl_sparse_?_trsm` routines support these configurations:

Column-major dense matrix: $layout =$ SPARSE_LAYOUT_COLUMN_MAJ OR	Row-major dense matrix: $layout =$ SPARSE_LAYOUT_ROW_MAJOR
--	---

0-based sparse matrix: SPARSE_INDEX_BASE_ZERO	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: SPARSE_INDEX_BASE_ONE	All formats	CSR BSR: general non-transposed matrix multiplication only

Input Parameters

<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NO Non-transpose, $op(A) = A$. N_TRANSPOSE SPARSE_OPERATION_TR Transpose, $op(A) = A^T$. ANSPOSE SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$. NJUGATE_TRANSPOSE
<i>alpha</i>	C_FLOAT for <code>mkl_sparse_s_trsm</code> C_DOUBLE for <code>mkl_sparse_d_trsm</code> C_FLOAT_COMPLEX for <code>mkl_sparse_c_trsm</code> C_DOUBLE_COMPLEX for <code>mkl_sparse_z_trsm</code> Specifies the scalar <i>alpha</i> .
<i>A</i>	SPARSE_MATRIX_T. Handle containing sparse matrix in internal data structure.
<i>descr</i>	MATRIX_DESCR. Descriptor specifying sparse matrix properties. <i>type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_ The matrix is processed as is. GENERAL SPARSE_MATRIX_TYPE_ The matrix is symmetric (only the requested SYMMETRIC triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is Hermitian (only the requested HERMITIAN triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is triangular (only the requested TRIANGULAR triangle is processed). SPARSE_MATRIX_TYPE_ The matrix is diagonal (only diagonal elements DIAGONAL are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). (Applies to BSR format only.)

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). (Applies to BSR format only.)

mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LO The lower triangular matrix part is processed.
WER

SPARSE_FILL_MODE_UP The upper triangular matrix part is processed.
PER

diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.
T

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

layout

C_INT.

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_STORAGE Storage of elements uses column major layout.
N_MAJOR

SPARSE_LAYOUT_ROW_STORAGE Storage of elements uses row major layout.
A_JOR

x

C_FLOAT for mkl_sparse_s_trsm

C_DOUBLE for mkl_sparse_d_trsm

C_FLOAT_COMPLEX for mkl_sparse_c_trsm

C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Array of size at least *rows*cols*.

	<i>layout</i> =	<i>layout</i> =
	SPARSE_LAYOUT_COLUMN_MAJOR	SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	number of rows in A
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

On entry, the array *x* must contain the matrix *x*.

columns

C_INT.

Number of columns in matrix *y*.

ldx

C_INT.

Specifies the leading dimension of matrix *x*.

y C_FLOAT for mkl_sparse_s_trsm
 C_DOUBLE for mkl_sparse_d_trsm
 C_FLOAT_COMPLEX for mkl_sparse_c_trsm
 C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Array of size at least *rows*cols*, where

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>y</i>)	<i>ldy</i>	number of rows in A
<i>cols</i> (number of columns in <i>y</i>)	<i>columns</i>	<i>ldy</i>

Output Parameters

y C_FLOAT for mkl_sparse_s_trsm
 C_DOUBLE for mkl_sparse_d_trsm
 C_FLOAT_COMPLEX for mkl_sparse_c_trsm
 C_DOUBLE_COMPLEX for mkl_sparse_z_trsm

Overwritten by the updated matrix *y*.

stat INTEGER.

Value indicating whether the operation was successful or not, and why:

- SPARSE_STATUS_SUCCESS The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.
- SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
- SPARSE_STATUS_EXECUTION_FAILED Execution failed.
- SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
- SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_?_add

Computes sum of two sparse matrices.

Syntax

```
stat = mkl_sparse_s_add (operation, A, alpha, B, C)
stat = mkl_sparse_d_add (operation, A, alpha, B, C)
stat = mkl_sparse_c_add (operation, A, alpha, B, C)
stat = mkl_sparse_z_add (operation, A, alpha, B, C)
```

Include Files

- `mkl_spblas.f90`

Description

The `mkl_sparse_?_add` routine performs a matrix-matrix operation:

```
 $C := \alpha * \text{op}(A) + B$ 
```

where *alpha* is a scalar and *A*, *B*, and *C* are sparse matrices.

Input Parameters

<i>A</i>	SPARSE_MATRIX_T. Handle containing a sparse matrix in internal data structure.
<i>alpha</i>	C_FLOAT for <code>mkl_sparse_s_add</code> C_DOUBLE for <code>mkl_sparse_d_add</code> C_FLOAT_COMPLEX for <code>mkl_sparse_c_add</code> C_DOUBLE_COMPLEX for <code>mkl_sparse_z_add</code> Specifies the scalar <i>alpha</i> .
<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NO Non-transpose, $\text{op}(A) = A$. N_TRANSPOSE SPARSE_OPERATION_TR Transpose, $\text{op}(A) = A^T$. ANSPOSE SPARSE_OPERATION_CO Conjugate transpose, $\text{op}(A) = A^H$. NJUGATE_TRANSPOSE
<i>B</i>	SPARSE_MATRIX_T. Handle containing a sparse matrix in internal data structure.

Output Parameters

<i>C</i>	SPARSE_MATRIX_T. Handle containing the resulting sparse matrix in internal data structure.
<i>stat</i>	INTEGER. Value indicating whether the operation was successful or not, and why:

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_spmv

Computes the product of two sparse matrices and stores the result as a sparse matrix.

Syntax

```
stat = mkl_sparse_spmv (operation, A, B, C)
```

Include Files

- mkl_spblas.f90

Description

The `mkl_sparse_spmv` routine performs a matrix-matrix operation:

```
C := op(A) * B
```

where A , B , and C are sparse matrices.

Input Parameters

<i>operation</i>	C_INT. Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NO_TRANSPOSE Non-transpose, $op(A) = A$. SPARSE_OPERATION_TRANSPOSE Transpose, $op(A) = A^T$. SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $op(A) = A^H$.
<i>A</i>	SPARSE_MATRIX_T. Handle containing a sparse matrix in internal data structure.
<i>B</i>	SPARSE_MATRIX_T.

Handle containing a sparse matrix in internal data structure.

Output Parameters

<i>C</i>	SPARSE_MATRIX_T. Handle containing the resulting sparse matrix in internal data structure.
<i>stat</i>	INTEGER. Value indicating whether the operation was successful or not, and why: SPARSE_STATUS_SUCCESS The operation was successful. SS SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array. SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed. _FAILED SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value. ID_VALUE SPARSE_STATUS_EXECUTION_FAILED Execution failed. TION_FAILED SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred. NAL_ERROR SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported. UPPORTED

mkl_sparse_?_spmmd

Computes the product of two sparse matrices and stores the result as a dense matrix.

Syntax

```
stat = mkl_sparse_s_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_d_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_c_spmmd (operation, A, B, layout, C, ldc)
stat = mkl_sparse_z_spmmd (operation, A, B, layout, C, ldc)
```

Include Files

- mkl_splblas.f90

Description

The `mkl_sparse_?_spmmd` routine performs a matrix-matrix operation:

$$C := \text{op}(A) * B$$

where *A* and *B* are sparse matrices and *C* is a dense matrix.

Input Parameters

<i>operation</i>	<p>C_INT.</p> <p>Specifies operation $op()$ on input matrix.</p> <p>SPARSE_OPERATION_NO Non-transpose, $op(A) = A$. N_TRANSPOSE</p> <p>SPARSE_OPERATION_TR Transpose, $op(A) = A^T$. ANSPOSE</p> <p>SPARSE_OPERATION_CO Conjugate transpose, $op(A) = A^H$. NJUGATE_TRANSPOSE</p>
<i>A</i>	<p>SPARSE_MATRIX_T.</p> <p>Handle containing a sparse matrix in internal data structure.</p>
<i>B</i>	<p>SPARSE_MATRIX_T.</p> <p>Handle containing a sparse matrix in internal data structure.</p>
<i>layout</i>	<p>C_INT.</p> <p>Describes the storage scheme for the dense matrix:</p> <p>SPARSE_LAYOUT_COLUMN Storage of elements uses column major layout. N_MAJOR</p> <p>SPARSE_LAYOUT_ROW_M Storage of elements uses row major layout. AJOR</p>
<i>ldC</i>	<p>C_INT.</p> <p>Leading dimension of matrix C.</p>

Output Parameters

<i>C</i>	<p>C_FLOAT for mkl_sparse_s_spmmd C_DOUBLE for mkl_sparse_d_spmmd C_FLOAT_COMPLEX for mkl_sparse_c_spmmd C_DOUBLE_COMPLEX for mkl_sparse_z_spmmd</p> <p>Resulting dense matrix stored in internal data structure.</p>
<i>stat</i>	<p>INTEGER.</p> <p>Value indicating whether the operation was successful or not, and why:</p> <p>SPARSE_STATUS_SUCCE The operation was successful. SS</p> <p>SPARSE_STATUS_NOT_I The routine encountered an empty handle or INITIALIZED matrix array.</p> <p>SPARSE_STATUS_ALLOC Internal memory allocation failed. _FAILED</p>

SPARSE_STATUS_INVALID_ID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

BLAS-like Extensions

Intel MKL provides C and Fortran routines to extend the functionality of the BLAS routines. These include routines to compute vector products, matrix-vector products, and matrix-matrix products.

Intel MKL also provides routines to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations. Transposition operations are Copy As Is, Conjugate transpose, Transpose, and Conjugate. Each routine adds the possibility of scaling during the transposition operation by giving some *alpha* and/or *beta* parameters. Each routine supports both row-major orderings and column-major orderings.

Table “BLAS-like Extensions” lists these routines.

The <?> symbol in the routine short names is a precision prefix that indicates the data type:

<i>s</i>	REAL
<i>d</i>	DOUBLE PRECISION
<i>c</i>	COMPLEX
<i>z</i>	DOUBLE COMPLEX

BLAS-like Extensions

Routine	Data Types	Description
?axpby	s, d, c, z	Scales two vectors, adds them to one another and stores result in the vector (routines)
?gem2vu	s, d	Two matrix-vector products using a general matrix, real data
?gem2vc	c, z	Two matrix-vector products using a general matrix, complex data
?gemm3m	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
?gemm3m_batch	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
mkl_?imatcopy	s, d, c, z	Performs scaling and in-place transposition/copying of matrices.
mkl_?omatcopy	s, d, c, z	Performs scaling and out-of-place transposition/copying of matrices.

Routine	Data Types	Description
<code>mkl_?omatcopy2</code>	s, d, c, z	Performs two-strided scaling and out-of-place transposition/copying of matrices.
<code>mkl_?omatadd</code>	s, d, c, z	Performs scaling and sum of two matrices including their out-of-place transposition/copying.

?axpby

Scales two vectors, adds them to one another and stores result in the vector.

Syntax

```
call saxpby(n, a, x, incx, b, y, incy)
call daxpby(n, a, x, incx, b, y, incy)
call caxpby(n, a, x, incx, b, y, incy)
call zaxpby(n, a, x, incx, b, y, incy)
call axpby(x, y [,a] [,b])
```

Include Files

- `mkl.fi, blas.f90`

Description

The ?axpby routines perform a vector-vector operation defined as

$$y := a*x + b*y$$

where:

a and *b* are scalars

x and *y* are vectors each with *n* elements.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>a</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Specifies the scalar <i>a</i> .
<i>x</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Array, size at least $(1 + (n-1)*abs(incx))$.

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>b</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Specifies the scalar <i>b</i> .
<i>y</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Array, size at least $(1 + (n-1)*abs(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>y</i>	Contains the updated vector <i>y</i> .
----------	--

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `axpby` interface are the following:

<i>x</i>	Holds the array of size <i>n</i> .
<i>y</i>	Holds the array of size <i>n</i> .
<i>a</i>	The default value is 1.
<i>b</i>	The default value is 1.

?gem2vu

Computes two matrix-vector products using a general matrix (real data)

Syntax

```
call sgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call dgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call gem2vu(a, x1, x2, y1, y2 [,alpha][,beta] )
```

Include Files

- `mk1.fi, blas.f90`

Description

The ?gem2vu routines perform two matrix-vector operations defined as

$$y1 := \alpha * A * x1 + \beta * y1,$$

and

$$y2 := \alpha * A' * x2 + \beta * y2,$$

where:

alpha and *beta* are scalars,

x1, *x2*, *y1*, and *y2* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.
<i>x1</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least $(1 + (n-1) * \text{abs}(\text{incx1}))$. Before entry, the incremented array <i>x1</i> must contain the vector <i>x1</i> .
<i>incx1</i>	INTEGER. Specifies the increment for the elements of <i>x1</i> . The value of <i>incx1</i> must not be zero.
<i>x2</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least $(1 + (m-1) * \text{abs}(\text{incx2}))$. Before entry, the incremented array <i>x2</i> must contain the vector <i>x2</i> .
<i>incx2</i>	INTEGER. Specifies the increment for the elements of <i>x2</i> . The value of <i>incx2</i> must not be zero.
<i>beta</i>	REAL for sgem2vu

	DOUBLE PRECISION for <code>dgem2vu</code>
	Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>y1</i> and <i>y2</i> need not be set on input.
<i>y1</i>	REAL for <code>sgem2vu</code> DOUBLE PRECISION for <code>dgem2vu</code>
	Array, size at least $(1 + (m-1) * \text{abs}(incy1))$. Before entry with non-zero <i>beta</i> , the incremented array <i>y1</i> must contain the vector <i>y1</i> .
<i>incy1</i>	INTEGER. Specifies the increment for the elements of <i>y1</i> . The value of <i>incy1</i> must not be zero.
<i>y</i>	REAL for <code>sgem2vu</code> DOUBLE PRECISION for <code>dgem2vu</code>
	Array, size at least $(1 + (n-1) * \text{abs}(incy2))$. Before entry with non-zero <i>beta</i> , the incremented array <i>y2</i> must contain the vector <i>y2</i> .
<i>incy2</i>	INTEGER. Specifies the increment for the elements of <i>y2</i> . The value of <i>incy2</i> must not be zero.

Output Parameters

<i>y1</i>	Updated vector <i>y1</i> .
<i>y2</i>	Updated vector <i>y2</i> .

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gem2vu` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>x1</i>	Holds the vector with the number of elements <i>rx1</i> where $rx1 = n$.
<i>x2</i>	Holds the vector with the number of elements <i>rx2</i> where $rx2 = m$.
<i>y1</i>	Holds the vector with the number of elements <i>ry1</i> where $ry1 = m$.
<i>y2</i>	Holds the vector with the number of elements <i>ry2</i> where $ry2 = n$.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?gem2vc

Computes two matrix-vector products using a general matrix (complex data)

Syntax

```
call cgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call zgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call gem2vc(a, x1, x2, y1, y2 [,alpha][,beta] )
```

Include Files

- mkl.fi, blas.f90

Description

The ?gem2vc routines perform two matrix-vector operations defined as

$$y1 := \alpha * A * x1 + \beta * y1,$$

and

$$y2 := \alpha * \text{conjg}(A') * x2 + \beta * y2,$$

where:

alpha and *beta* are scalars,

x1, *x2*, *y1*, and *y2* are vectors,

A is an *m*-by-*n* matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.
<i>x1</i>	COMPLEX for cgem2vc DOUBLE COMPLEX for zgem2vc Array, size at least $(1 + (n-1) * \text{abs}(\text{incx1}))$. Before entry, the incremented array <i>x1</i> must contain the vector <i>x1</i> .
<i>incx1</i>	INTEGER. Specifies the increment for the elements of <i>x1</i> . The value of <i>incx1</i> must not be zero.

<i>x2</i>	COMPLEX for <i>cgem2vc</i> DOUBLE COMPLEX for <i>zgem2vc</i> Array, size at least $(1+(m-1)*abs(incx2))$. Before entry, the incremented array <i>x2</i> must contain the vector <i>x2</i> .
<i>incx2</i>	INTEGER. Specifies the increment for the elements of <i>x2</i> . The value of <i>incx2</i> must not be zero.
<i>beta</i>	COMPLEX for <i>cgem2vc</i> DOUBLE COMPLEX for <i>zgem2vc</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>y1</i> and <i>y2</i> need not be set on input.
<i>y1</i>	COMPLEX for <i>cgem2vc</i> DOUBLE COMPLEX for <i>zgem2vc</i> Array, size at least $(1+(m-1)*abs(incy1))$. Before entry with non-zero <i>beta</i> , the incremented array <i>y1</i> must contain the vector <i>y1</i> .
<i>incy1</i>	INTEGER. Specifies the increment for the elements of <i>y1</i> . The value of <i>incy1</i> must not be zero.
<i>y2</i>	COMPLEX for <i>cgem2vc</i> DOUBLE COMPLEX for <i>zgem2vc</i> Array, size at least $(1+(n-1)*abs(incy2))$. Before entry with non-zero <i>beta</i> , the incremented array <i>y2</i> must contain the vector <i>y2</i> .
<i>incy2</i>	INTEGER. Specifies the increment for the elements of <i>y2</i> . The value of <i>incy</i> must not be zero. INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>y1</i>	Updated vector <i>y1</i> .
<i>y2</i>	Updated vector <i>y2</i> .

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *gem2vc* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>x1</i>	Holds the vector with the number of elements <i>rx1</i> where $rx1 = n$.
<i>x2</i>	Holds the vector with the number of elements <i>rx2</i> where $rx2 = m$.

<i>y1</i>	Holds the vector with the number of elements <i>ry1</i> where $ry1 = m$.
<i>y2</i>	Holds the vector with the number of elements <i>ry2</i> where $ry2 = n$.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?gemmt

Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.

Syntax

```
call sgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemmt (uplo, transa, transb, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemmt (a, b, c[, uplo] [, transa] [, transb] [, alpha] [, beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?gemmt routines compute a scalar-matrix-matrix product with general matrices and add the result to the upper or lower part of a scalar-matrix product. These routines are similar to the ?gemm routines, but they only access and update a triangular part of the square result matrix (see Application Notes below).

The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

alpha and *beta* are scalars,

A, *B* and *C* are matrices:

$\text{op}(A)$ is an *n*-by-*k* matrix,

$\text{op}(B)$ is a *k*-by-*n* matrix,

C is an *n*-by-*n* upper or lower triangular matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array <i>c</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication:

	<p>if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$;</p> <p>if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A^T$;</p> <p>if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = A^H$.</p>
<i>transb</i>	<p>CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication:</p> <p>if <i>transb</i> = 'N' or 'n', then $\text{op}(B) = B$;</p> <p>if <i>transb</i> = 'T' or 't', then $\text{op}(B) = B^T$;</p> <p>if <i>transb</i> = 'C' or 'c', then $\text{op}(B) = B^H$.</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>sgemmt</i></p> <p>DOUBLE PRECISION for <i>dgemmt</i></p> <p>COMPLEX for <i>cgemmt</i></p> <p>DOUBLE COMPLEX for <i>zgemmt</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <i>sgemmt</i></p> <p>DOUBLE PRECISION for <i>dgemmt</i></p> <p>COMPLEX for <i>cgemmt</i></p> <p>DOUBLE COMPLEX for <i>zgemmt</i></p> <p>Array, size <i>lda</i> by <i>ka</i>, where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>b</i>	<p>REAL for <i>sgemmt</i></p> <p>DOUBLE PRECISION for <i>dgemmt</i></p> <p>COMPLEX for <i>cgemmt</i></p> <p>DOUBLE COMPLEX for <i>zgemmt</i></p> <p>Array, size <i>ldb</i> by <i>kb</i>, where <i>kb</i> is <i>n</i> when <i>transb</i> = 'N' or 'n', and is <i>k</i> otherwise. Before entry with <i>transb</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>n</i>-by-<i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>

<i>ldb</i>	<p>INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When <i>transb</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, k)$, otherwise <i>ldb</i> must be at least $\max(1, n)$.</p>
<i>beta</i>	<p>REAL for <i>sgemmt</i> DOUBLE PRECISION for <i>dgemmt</i> COMPLEX for <i>cgemmt</i> DOUBLE COMPLEX for <i>zgemmt</i></p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is equal to zero, then <i>c</i> need not be set on input.</p>
<i>c</i>	<p>REAL for <i>sgemmt</i> DOUBLE PRECISION for <i>dgemmt</i> COMPLEX for <i>cgemmt</i> DOUBLE COMPLEX for <i>zgemmt</i></p> <p>Array, size <i>ldc</i> by <i>n</i>.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the matrix <i>C</i> and the strictly lower triangular part of <i>c</i> is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the matrix <i>C</i> and the strictly upper triangular part of <i>c</i> is not referenced.</p> <p>When <i>beta</i> is equal to zero, then <i>c</i> need not be set on input.</p>
<i>ldc</i>	<p>INTEGER. Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program. The value of <i>ldc</i> must be at least $\max(1, n)$.</p>

Output Parameters

<i>c</i>	<p>When <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>c</i> is overwritten by the upper triangular part of the updated matrix.</p> <p>When <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>c</i> is overwritten by the lower triangular part of the updated matrix.</p>
----------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gemmt* interface are the following:

<i>a</i>	<p>Holds the matrix <i>A</i> of size (<i>ma</i>,<i>ka</i>) where</p> <p><i>ka</i> = <i>k</i> if <i>transa</i>='N', <i>ka</i> = <i>n</i> otherwise, <i>ma</i> = <i>n</i> if <i>transa</i>='N', <i>ma</i> = <i>k</i> otherwise.</p>
----------	--

<i>b</i>	Holds the matrix <i>B</i> of size (<i>mb</i> , <i>kb</i>) where $kb = n$ if <i>transb</i> = 'N', $kb = k$ otherwise, $mb = k$ if <i>transb</i> = 'N', $mb = n$ otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>transb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

Application Notes

These routines only access and update the upper or lower triangular part of the result matrix. This can be useful when the result is known to be symmetric; for example, when computing a product of the form $C := \alpha * B * S * B^T + \beta * C$, where *S* and *C* are symmetric matrices and *B* is a general matrix. In this case, first compute $A := B * S$ (which can be done using the corresponding ?*symm* routine), then compute $C := \alpha * A * B^T + \beta * C$ using the ?*gemmt* routine.

?gemm3m

Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.

Syntax

```
call cgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call gemm3m(a, b, c [,transa][,transb] [,alpha][,beta])
```

Include Files

- mkl.fi, blas.f90

Description

The ?*gemm3m* routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the ?*gemm* routines, but they use fewer matrix multiplication operations (see *Application Notes* below).

The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$, or $\text{op}(x) = \text{conjg}(x')$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>transa</i>	<p>CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication:</p> <p>if <i>transa</i> = 'N' or 'n', then $\text{op}(A) = A$;</p> <p>if <i>transa</i> = 'T' or 't', then $\text{op}(A) = A'$;</p> <p>if <i>transa</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A')$.</p>
<i>transb</i>	<p>CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication:</p> <p>if <i>transb</i> = 'N' or 'n', then $\text{op}(B) = B$;</p> <p>if <i>transb</i> = 'T' or 't', then $\text{op}(B) = B'$;</p> <p>if <i>transb</i> = 'C' or 'c', then $\text{op}(B) = \text{conjg}(B')$.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C. The value of <i>m</i> must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C.</p> <p>The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>COMPLEX for <i>cgemm3m</i></p> <p>DOUBLE COMPLEX for <i>zgemm3m</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>COMPLEX for <i>cgemm3m</i></p> <p>DOUBLE COMPLEX for <i>zgemm3m</i></p> <p>Array, size <i>lda</i> by <i>ka</i>, where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix A, otherwise the leading <i>k</i>-by-<i>m</i> part of the array <i>a</i> must contain the matrix A.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program.</p>

When $transa = 'N'$ or $'n'$, then lda must be at least $\max(1, m)$, otherwise lda must be at least $\max(1, k)$.

b	<p>COMPLEX for <code>cgemm3m</code></p> <p>DOUBLE COMPLEX for <code>zgemm3m</code></p> <p>Array, size ldb by kb, where kb is n when $transa = 'N'$ or $'n'$, and is k otherwise. Before entry with $transa = 'N'$ or $'n'$, the leading k-by-n part of the array b must contain the matrix B, otherwise the leading n-by-k part of the array b must contain the matrix B.</p>
ldb	<p>INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program.</p> <p>When $transa = 'N'$ or $'n'$, then ldb must be at least $\max(1, k)$, otherwise ldb must be at least $\max(1, n)$.</p>
$beta$	<p>COMPLEX for <code>cgemm3m</code></p> <p>DOUBLE COMPLEX for <code>zgemm3m</code></p> <p>Specifies the scalar $beta$.</p> <p>When $beta$ is equal to zero, then c need not be set on input.</p>
c	<p>COMPLEX for <code>cgemm3m</code></p> <p>DOUBLE COMPLEX for <code>zgemm3m</code></p> <p>Array, size ldc by n. Before entry, the leading m-by-n part of the array c must contain the matrix C, except when $beta$ is equal to zero, in which case c need not be set on entry.</p>
ldc	<p>INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program.</p> <p>The value of ldc must be at least $\max(1, m)$.</p>

Output Parameters

c	Overwritten by the m -by- n matrix $(\alpha * op(A) * op(B) + \beta * C)$.
-----	---

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine `gemm3m` interface are the following:

a	<p>Holds the matrix A of size (ma,ka) where</p> <p>$ka = k$ if $transa='N'$,</p> <p>$ka = m$ otherwise,</p> <p>$ma = m$ if $transa='N'$,</p> <p>$ma = k$ otherwise.</p>
b	Holds the matrix B of size (mb,kb) where

$kb = n$ if $transb = 'N'$,
 $kb = k$ otherwise,
 $mb = k$ if $transb = 'N'$,
 $mb = n$ otherwise.

c Holds the matrix C of size (m,n) .
 $transa$ Must be 'N', 'C', or 'T'.
 The default value is 'N'.
 $transb$ Must be 'N', 'C', or 'T'.
 The default value is 'N'.
 $alpha$ The default value is 1.
 $beta$ The default value is 1.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$fl(x \text{ op } y) = (x \text{ op } y) (1 + \delta), \quad |\delta| \leq u, \quad \text{op} = *, /, \quad fl(x \pm y) = x(1 + \alpha) \pm y(1 + \beta), \quad |\alpha|, |\beta| \leq u$$

then for an n -by- n matrix $\hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$, the following bounds are satisfied:

$$\|\hat{C}_1 - C_1\| \leq 2(n+1)u \|A\|_\infty \|B\|_\infty + O(u^2),$$

$$\|\hat{C}_2 - C_2\| \leq 4(n+4)u \|A\|_\infty \|B\|_\infty + O(u^2),$$

where $\|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty)$, and $\|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

?gemm3m_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

```
call cgemm3m_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call zgemm3m_batch(transa_array, transb_array, m_array, n_array, k_array, alpha_array,
a_array, lda_array, b_array, ldb_array, beta_array, c_array, ldc_array, group_count,
group_size)
```

```
call gemm3m_batch(a_array, b_array, c_array, group_size [,transa_array][,transb_array]
[,alpha_array][,beta_array])
```

Include Files

- `mkl.fi`, `blas.f90`

Description

The `?gemm3m_batch` routines perform a series of matrix-matrix operations with general matrices. They are similar to the `?gemm3m` routine counterparts, but the `?gemm3m_batch` routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters. The `?gemm3m_batch` routines use fewer matrix multiplications than the `?gemm_batch` routines, as described in the *Application Notes*.

The operation is defined as

```
idx = 1
for i = 1..group_count
  alpha and beta in alpha_array(i) and beta_array(i)
  for j = 1..group_size(i)
    A, B, and C matrix in a_array(idx), b_array(idx), and c_array(idx)
    C := alpha*op(A)*op(B) + beta*C,
    idx = idx + 1
  end for
end for
```

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

alpha and *beta* are scalar elements of *alpha_array* and *beta_array*,

A, *B* and *C* are matrices such that for *m*, *n*, and *k* which are elements of *m_array*, *n_array*, and *k_array*:

$\text{op}(A)$ is an *m*-by-*k* matrix,

$\text{op}(B)$ is a *k*-by-*n* matrix,

C is an *m*-by-*n* matrix.

A, *B*, and *C* represent matrices stored at addresses pointed to by *a_array*, *b_array*, and *c_array*, respectively. The number of entries in *a_array*, *b_array*, and *c_array* is *total_batch_count* = the sum of all the *group_size* entries.

See also [gemm](#) for a detailed description of multiplication for general matrices and [gemm_batch](#), BLAS-like extension routines for similar matrix-matrix operations.

NOTE

Error checking is not performed for Intel MKL Windows* single dynamic libraries for the `?gemm3m_batch` routines.

Input Parameters

transa_array CHARACTER*1. Array of size *group_count*. For the group *i*, *transa_i* = *transa_array*(*i*) specifies the form of $\text{op}(A)$ used in the matrix multiplication:

- if *transa_i* = 'N' or 'n', then $\text{op}(A) = A$;
- if *transa_i* = 'T' or 't', then $\text{op}(A) = A^T$;
- if *transa_i* = 'C' or 'c', then $\text{op}(A) = A^H$.

<i>transb_array</i>	<p>CHARACTER*1. Array of size <i>group_count</i>. For the group <i>i</i>, <i>transb_i</i> = <i>transb_array(i)</i> specifies the form of $\text{op}(B_i)$ used in the matrix multiplication:</p> <p>if <i>transb_i</i> = 'N' or 'n', then $\text{op}(B) = B$;</p> <p>if <i>transb_i</i> = 'T' or 't', then $\text{op}(B) = B^T$;</p> <p>if <i>transb_i</i> = 'C' or 'c', then $\text{op}(B) = B^H$.</p>
<i>m_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>m_i</i> = <i>m_array(i)</i> specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix <i>C</i>.</p> <p>The value of each element of <i>m_array</i> must be at least zero.</p>
<i>n_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>n_i</i> = <i>n_array(i)</i> specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix <i>C</i>.</p> <p>The value of each element of <i>n_array</i> must be at least zero.</p>
<i>k_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>k_i</i> = <i>k_array(i)</i> specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$.</p> <p>The value of each element of <i>k_array</i> must be at least zero.</p>
<i>alpha_array</i>	<p>REAL for <i>sgemm_batch</i></p> <p>DOUBLE PRECISION for <i>dgemm_batch</i></p> <p>COMPLEX for <i>cgemm_batch</i>, <i>scgemm_batch</i></p> <p>DOUBLE COMPLEX for <i>zgemm_batch</i>, <i>dzgemm_batch</i></p> <p>Array of size <i>group_count</i>. For the group <i>i</i>, <i>alpha_array(i)</i> specifies the scalar <i>alpha_i</i>.</p>
<i>a_array</i>	<p>INTEGER*8 for Intel® 64 architecture</p> <p>INTEGER*4 for IA-32 architecture</p> <p>Array, size <i>total_batch_count</i>, of pointers to arrays used to store <i>A</i> matrices.</p>
<i>lda_array</i>	<p>INTEGER. Array of size <i>group_count</i>. For the group <i>i</i>, <i>lda_i</i> = <i>lda_array(i)</i> specifies the leading dimension of the array storing matrix <i>A</i> as declared in the calling (sub)program.</p> <p>When <i>transa_i</i> = 'N' or 'n', then <i>lda_i</i> must be at least $\max(1, m_i)$, otherwise <i>lda_i</i> must be at least $\max(1, k_i)$.</p>
<i>b_array</i>	<p>INTEGER*8 for Intel® 64 architecture</p> <p>INTEGER*4 for IA-32 architecture</p> <p>Array, size <i>total_batch_count</i>, of pointers to arrays used to store <i>B</i> matrices.</p>
<i>ldb_array</i>	<p>INTEGER.</p>

Array of size *group_count*. For the group *i*, $ldb_i = ldb_array(i)$ specifies the leading dimension of the array storing matrix *B* as declared in the calling (sub)program.

When $transb_i = 'N'$ or $'n'$, then ldb_i must be at least $\max(1, k_i)$, otherwise ldb_i must be at least $\max(1, n_i)$.

beta_array REAL for *sgemm_batch*
 DOUBLE PRECISION for *dgemm_batch*
 COMPLEX for *cgemm_batch*, *scgemm_batch*
 DOUBLE COMPLEX for *zgemm_batch*, *dzgemm_batch*
 For the group *i*, *beta_array(i)* specifies the scalar *beta_i*.
 When *beta_i* is equal to zero, then *C* matrices in group *i* need not be set on input.

c_array INTEGER*8 for Intel® 64 architecture
 INTEGER*4 for IA-32 architecture
 Array, size *total_batch_count*, of pointers to arrays used to store *C* matrices.

ldc_array INTEGER.
 Array of size *group_count*. For the group *i*, $ldc_i = ldc_array(i)$ specifies the leading dimension of all arrays storing matrix *C* in group *i* as declared in the calling (sub)program.
 ldc_i must be at least $\max(1, m_i)$.

group_count INTEGER.
 Specifies the number of groups. Must be at least 0.

group_size INTEGER.
 Array of size *group_count*. The element *group_size(i)* specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array Overwritten by the m_i -by- n_i matrix $(\alpha_i * op(A) * op(B) + \beta_i * C)$ for group *i*.

BLAS 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [BLAS 95 Interface Conventions](#).

Specific details for the routine *gemm3m_batch* interface are the following:

a_array Holds the matrix *A* of size (m_a, k_a) where
 $k_a = k$ if $transa = 'N'$,
 $k_a = m$ otherwise,

$ma = m$ if $transa='N'$,
 $ma = k$ otherwise.

b_array Holds the matrix B of size (mb, kb) where
 $kb = n$ if $transb_array = 'N'$,
 $kb = k$ otherwise,
 $mb = k$ if $transb_array = 'N'$,
 $mb = n$ otherwise.

c_array Holds the matrix C of size (m, n) .

transa_array Must be 'N', 'C', or 'T'.
The default value is 'N'.

transb_array Must be 'N', 'C', or 'T'.
The default value is 'N'.

alpha_array The default value is 1.

beta_array The default value is 0.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$fl(x \text{ op } y) = (x \text{ op } y) (1 + \delta), \quad |\delta| \leq u, \quad \text{op} = *, /, \quad fl(x \pm y) = x(1 + \alpha) \pm y(1 + \beta), \quad |\alpha|, |\beta| \leq u$$

then for an n -by- n matrix $\hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$, the following bounds are satisfied:

$$\|\hat{C}_1 - C_1\| \leq 2(n+1)u \|A\|_\infty \|B\|_\infty + O(u^2),$$

$$\|\hat{C}_2 - C_2\| \leq 4(n+4)u \|A\|_\infty \|B\|_\infty + O(u^2),$$

where $\|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty)$, and $\|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

mkl_?imatcopy

Performs scaling and in-place transposition/copying of matrices.

Syntax

call `mkl_simatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)`

call `mkl_dimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)`

call `mkl_cimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)`

call `mkl_zimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)`

Include Files

- `mkl.fi`

Description

The `mkl_?imatcopy` routine performs scaling and in-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$AB := \alpha * op(AB).$$

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $op(AB) = AB$ and the matrix <i>AB</i> is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that <i>AB</i> should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that <i>AB</i> should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that <i>AB</i> should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	INTEGER. The number of rows in matrix <i>AB</i> before the transpose operation.
<i>cols</i>	INTEGER. The number of columns in matrix <i>AB</i> before the transpose operation.
<i>ab</i>	REAL for <code>mkl_simatcopy</code> . DOUBLE PRECISION for <code>mkl_dimatcopy</code> . COMPLEX for <code>mkl_cimatcopy</code> . DOUBLE COMPLEX for <code>mkl_zimatcopy</code> . Array, size $ab(lda, *)$.
<i>alpha</i>	REAL for <code>mkl_simatcopy</code> . DOUBLE PRECISION for <code>mkl_dimatcopy</code> . COMPLEX for <code>mkl_cimatcopy</code> .

DOUBLE COMPLEX for `mkl_zimatcopy`.

This parameter scales the input matrix by *alpha*.

lda INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.

This parameter must be at least *rows* if *ordering* = 'C' or 'c', and $\max(1, cols)$ otherwise.

ldb INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.

To determine the minimum value of *ldb* on output, consider the following guideline:

If *ordering* = 'C' or 'c', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, cols)$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, rows)$

If *ordering* = 'R' or 'r', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, rows)$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, cols)$

Output Parameters

ab REAL for `mkl_simatcopy`.
 DOUBLE PRECISION for `mkl_dimatcopy`.
 COMPLEX for `mkl_cimatcopy`.
 DOUBLE COMPLEX for `mkl_zimatcopy`.
 Array, size `ab(ldb,*)`.
 Contains the matrix *AB*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_simatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  REAL ab(*), alpha*
```

```
SUBROUTINE mkl_dimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  DOUBLE PRECISION ab(*), alpha*
```

```

SUBROUTINE mkl_cimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  COMPLEX ab(*), alpha*

```

```

SUBROUTINE mkl_zimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  DOUBLE COMPLEX ab(*), alpha*

```

mkl_?omatcopy

Performs scaling and out-place transposition/copying of matrices.

Syntax

```

call mkl_somatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_domatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_comatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_zomatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)

```

Include Files

- mkl.fi

Description

The `mkl_?omatcopy` routine performs scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \alpha * \text{op}(A)$$

The routine parameter descriptions are common for all implemented interfaces with the exception of data types that mostly refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $\text{op}(A) = A$ and the matrix <i>A</i> is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that <i>A</i> should be transposed.

If $trans = 'C'$ or $'c'$, it is assumed that A should be conjugate transposed.

If $trans = 'R'$ or $'r'$, it is assumed that A should be only conjugated.

If the data is real, then $trans = 'R'$ is the same as $trans = 'N'$, and $trans = 'C'$ is the same as $trans = 'T'$.

<i>rows</i>	INTEGER. The number of rows in the source matrix.
<i>cols</i>	INTEGER. The number of columns in the source matrix.
<i>alpha</i>	<p>REAL for <code>mkl_somatcopy</code>.</p> <p>DOUBLE PRECISION for <code>mkl_domatcopy</code>.</p> <p>COMPLEX for <code>mkl_comatcopy</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zomatcopy</code>.</p> <p>This parameter scales the input matrix by <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <code>mkl_somatcopy</code>.</p> <p>DOUBLE PRECISION for <code>mkl_domatcopy</code>.</p> <p>COMPLEX for <code>mkl_comatcopy</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zomatcopy</code>.</p> <p>Array, size $a(lda, *)$.</p>
<i>lda</i>	<p>INTEGER. (Fortran interface). Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.</p> <p>This parameter must be at least $\max(1, rows)$ if $ordering = 'C'$ or $'c'$, and $\max(1, cols)$ otherwise.</p>
<i>b</i>	<p>REAL for <code>mkl_somatcopy</code>.</p> <p>DOUBLE PRECISION for <code>mkl_domatcopy</code>.</p> <p>COMPLEX for <code>mkl_comatcopy</code>.</p> <p>DOUBLE COMPLEX for <code>mkl_zomatcopy</code>.</p> <p>Array, size $b(l db, *)$.</p>
<i>ldb</i>	<p>INTEGER. (Fortran interface). Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.</p> <p>To determine the minimum value of <i>ldb</i> on output, consider the following guideline:</p> <p>If $ordering = 'C'$ or $'c'$, then</p> <ul style="list-style-type: none"> • If $trans = 'T'$ or $'t'$ or $'C'$ or $'c'$, this parameter must be at least $\max(1, cols)$ • If $trans = 'N'$ or $'n'$ or $'R'$ or $'r'$, this parameter must be at least $\max(1, rows)$

If *ordering* = 'R' or 'r', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

Output Parameters

b REAL for `mkl_somatcopy`.
 DOUBLE PRECISION for `mkl_domatcopy`.
 COMPLEX for `mkl_comatcopy`.
 DOUBLE COMPLEX for `mkl_zomatcopy`.
 Array, size at least *m*.
 Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  REAL alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_domatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  DOUBLE PRECISION alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_comatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  COMPLEX alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_zomatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  DOUBLE COMPLEX alpha, b(ldb,*), a(lda,*)
```

mkl_?omatcopy2

Performs two-strided scaling and out-of-place transposition/copying of matrices.

Syntax

```
call mkl_somatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb)
```

```
call mkl_domatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb)
```

```
call mkl_comatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb)
```

call `mkl_zomatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb)`

Include Files

- `mkl.fi`

Description

The `mkl_?omatcopy2` routine performs two-strided scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \alpha * op(A)$$

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, $A(2,1)$ is stored in memory one element away from $A(1,1)$, but $A(1,2)$ is a leading dimension away. The leading dimension in this case is the single stride. If a matrix has two strides, then both $A(2,1)$ and $A(1,2)$ may be an arbitrary distance from $A(1,1)$.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $op(A)=A$ and the matrix <i>A</i> is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that <i>A</i> should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that <i>A</i> should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that <i>A</i> should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	INTEGER. The number of matrix rows.
<i>cols</i>	INTEGER. The number of matrix columns.
<i>alpha</i>	REAL for <code>mkl_somatcopy2</code> . DOUBLE PRECISION for <code>mkl_domatcopy2</code> . COMPLEX for <code>mkl_comatcopy2</code> . DOUBLE COMPLEX for <code>mkl_zomatcopy2</code> .

This parameter scales the input matrix by *alpha*.

<i>a</i>	<p>REAL for <code>mkl_somatcopy2</code>. DOUBLE PRECISION for <code>mkl_domatcopy2</code>. COMPLEX for <code>mkl_comatcopy2</code>. DOUBLE COMPLEX for <code>mkl_zomatcopy2</code>.</p> <p>Array, size $a(*)$.</p>
<i>lda</i>	<p>INTEGER. Distance between the first elements in adjacent rows in the source matrix; measured in the number of elements.</p> <p>This parameter must be at least $\max(1, rows)$ if <i>ordering</i> = 'C' or 'c', and $\max(1, cols)$ otherwise.</p>
<i>stridea</i>	<p>INTEGER. Distance between the first elements in adjacent columns (for row-major layout) or between the first elements in adjacent rows (for column-major layout) in the source matrix; measured in the number of elements.</p> <p>This parameter must be at least $\max(1, cols)$.</p>
<i>b</i>	<p>REAL for <code>mkl_somatcopy2</code>. DOUBLE PRECISION for <code>mkl_domatcopy2</code>. COMPLEX for <code>mkl_comatcopy2</code>. DOUBLE COMPLEX for <code>mkl_zomatcopy2</code>.</p> <p>Array, size $b(*)$.</p>
<i>ldb</i>	<p>INTEGER. Distance between the first elements in adjacent rows in the destination matrix; measured in the number of elements.</p> <p>To determine the minimum value of <i>ldb</i> on output, consider the following guideline:</p> <ul style="list-style-type: none"> • If <i>trans</i> = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, rows)$ • If <i>trans</i> = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, cols)$
<i>strideb</i>	<p>INTEGER. Distance between the first elements in adjacent columns in the destination matrix; measured in the number of elements.</p> <p>To determine the minimum value of <i>strideb</i> on output, consider the following guideline:</p> <ul style="list-style-type: none"> • If <i>trans</i> = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, cols)$ • If <i>trans</i> = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, rows)$

Output Parameters

<i>b</i>	REAL for <code>mkl_somatcopy2</code> .
----------	--

DOUBLE PRECISION for mkl_domatcopy2.

COMPLEX for mkl_comatcopy2.

DOUBLE COMPLEX for mkl_zomatcopy2.

Array, size at least m .

Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  REAL alpha, b(*), a(*)
```

```
SUBROUTINE mkl_domatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  DOUBLE PRECISION alpha, b(*), a(*)
```

```
SUBROUTINE mkl_comatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  COMPLEX alpha, b(*), a(*)
```

```
SUBROUTINE mkl_zomatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  DOUBLE COMPLEX alpha, b(*), a(*)
```

mkl_?omatadd

Scales and sums two matrices including in addition to performing out-of-place transposition operations.

Syntax

call mkl_somatadd(*ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc*)

call mkl_domatadd(*ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc*)

call mkl_comatadd(*ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc*)

call mkl_zomatadd(*ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc*)

Include Files

- mkl.fi

Description

The `mkl_?omatadd` routine scales and adds two matrices, as well as performing out-of-place transposition operations. A transposition operation can be no operation, a transposition, a conjugate transposition, or a conjugation (without transposition). The following out-of-place memory movement is done:

$C := \alpha * \text{op}(A) + \beta * \text{op}(B)$

where the $\text{op}(A)$ and $\text{op}(B)$ operations are transpose, conjugate-transpose, conjugate (no transpose), or no transpose, depending on the values of *transa* and *transb*. If no transposition of the source matrices is required, *m* is the number of rows and *n* is the number of columns in the source matrices *A* and *B*. In this case, the output matrix *C* is *m*-by-*n*.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Note that different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>transa</i>	CHARACTER*1. Parameter that specifies the operation type on matrix <i>A</i> . If <i>transa</i> = 'N' or 'n', $\text{op}(A)=A$ and the matrix <i>A</i> is assumed unchanged on input. If <i>transa</i> = 'T' or 't', it is assumed that <i>A</i> should be transposed. If <i>transa</i> = 'C' or 'c', it is assumed that <i>A</i> should be conjugate transposed. If <i>transa</i> = 'R' or 'r', it is assumed that <i>A</i> should be conjugated (and not transposed). If the data is real, then <i>transa</i> = 'R' is the same as <i>transa</i> = 'N', and <i>transa</i> = 'C' is the same as <i>transa</i> = 'T'.
<i>transb</i>	CHARACTER*1. Parameter that specifies the operation type on matrix <i>B</i> . If <i>transb</i> = 'N' or 'n', $\text{op}(B)=B$ and the matrix <i>B</i> is assumed unchanged on input. If <i>transb</i> = 'T' or 't', it is assumed that <i>B</i> should be transposed. If <i>transb</i> = 'C' or 'c', it is assumed that <i>B</i> should be conjugate transposed. If <i>transb</i> = 'R' or 'r', it is assumed that <i>B</i> should be conjugated (and not transposed). If the data is real, then <i>transb</i> = 'R' is the same as <i>transb</i> = 'N', and <i>transb</i> = 'C' is the same as <i>transb</i> = 'T'.
<i>m</i>	INTEGER. The number of matrix rows in $\text{op}(A)$, $\text{op}(B)$, and <i>C</i> .
<i>n</i>	INTEGER. The number of matrix columns in $\text{op}(A)$, $\text{op}(B)$, and <i>C</i> .
<i>alpha</i>	REAL for mkl_somatadd. DOUBLE PRECISION for mkl_domatadd. COMPLEX for mkl_comatadd.

DOUBLE COMPLEX for `mkl_zomatadd`.

This parameter scales the input matrix by *alpha*.

a REAL for `mkl_somatadd`.
 DOUBLE PRECISION for `mkl_domatadd`.
 COMPLEX for `mkl_comatadd`.
 DOUBLE COMPLEX for `mkl_zomatadd`.
 Array, size `a(lda,*)`.

lda INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix *A*; measured in the number of elements.
 For *ordering* = 'C' or 'c': when *transa* = 'N', 'n', 'R', or 'r', *lda* must be at least $\max(1, m)$; otherwise *lda* must be $\max(1, n)$.
 For *ordering* = 'R' or 'r': when *transa* = 'N', 'n', 'R', or 'r', *lda* must be at least $\max(1, n)$; otherwise *lda* must be $\max(1, m)$.

beta REAL for `mkl_somatadd`.
 DOUBLE PRECISION for `mkl_domatadd`.
 COMPLEX for `mkl_comatadd`.
 DOUBLE COMPLEX for `mkl_zomatadd`.
 This parameter scales the input matrix by *beta*.

b REAL for `mkl_somatadd`.
 DOUBLE PRECISION for `mkl_domatadd`.
 COMPLEX for `mkl_comatadd`.
 DOUBLE COMPLEX for `mkl_zomatadd`.
 Array, size `b(ldb,*)`.

ldb INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix *B*; measured in the number of elements.
 For *ordering* = 'C' or 'c': when *transa* = 'N', 'n', 'R', or 'r', *ldb* must be at least $\max(1, m)$; otherwise *ldb* must be $\max(1, n)$.
 For *ordering* = 'R' or 'r': when *transa* = 'N', 'n', 'R', or 'r', *ldb* must be at least $\max(1, n)$; otherwise *ldb* must be $\max(1, m)$.

ldc INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix *C*; measured in the number of elements.
 If *ordering* = 'C' or 'c', then *ldc* must be at least $\max(1, m)$, otherwise *ldc* must be at least $\max(1, n)$.

Output Parameters

c REAL for `mkl_somatadd`.

DOUBLE PRECISION for mkl_domatadd.
COMPLEX for mkl_comatadd.
DOUBLE COMPLEX for mkl_zomatadd.
Array, size $c(ldc, *)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )  
  CHARACTER*1 ordering, transa, transb  
  INTEGER m, n, lda, ldb, ldc  
  REAL alpha, beta  
  REAL a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_domatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )  
  CHARACTER*1 ordering, transa, transb  
  INTEGER m, n, lda, ldb, ldc  
  DOUBLE PRECISION alpha, beta  
  DOUBLE PRECISION a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_comatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )  
  CHARACTER*1 ordering, transa, transb  
  INTEGER m, n, lda, ldb, ldc  
  COMPLEX alpha, beta  
  COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zomatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )  
  CHARACTER*1 ordering, transa, transb  
  INTEGER m, n, lda, ldb, ldc  
  DOUBLE COMPLEX alpha, beta  
  DOUBLE COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```


LAPACK Routines

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations, linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (full, packed, and rectangular full packed (RFP) storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (full, packed, and RFP storage)
- triangular banded
- tridiagonal
- diagonally dominant tridiagonal.

NOTE

Different arrays used as parameters to Intel MKL LAPACK routines must not overlap.

WARNING

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as `INF` or `NaN` values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Intel MKL supports the Fortran 95 interface, which uses simplified routine calls with shorter argument lists, in addition to the FORTRAN 77 interface to LAPACK computational and driver routines. The syntax section of the routine description gives the calling sequence for the Fortran 95 interface, where available, immediately after the FORTRAN 77 calls.

Routine Naming Conventions

To call one of the routines from a FORTRAN 77 program, you can use the LAPACK name.

LAPACK names have the structure `?yyzzz` or `?yzzz`, where the initial symbol `?` indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

Some routines can have combined character codes, such as `ds` or `zc`.

The Fortran 95 interfaces to the LAPACK computational and driver routines are the same as the FORTRAN 77 names but without the first letter that indicates the data type. For example, the name of the routine that performs a triangular factorization of general real matrices in Fortran 95 is `getrf`. Different data types are handled through the definition of a specific internal parameter that refers to a module block with named constants for single and double precision.

Fortran 95 Interface Conventions

Intel® MKL implements the Fortran 95 interface to LAPACK through wrappers that call respective FORTRAN 77 routines. This interface uses such Fortran 95 features as assumed-shape arrays and optional arguments to provide simplified calls to LAPACK routines with fewer arguments.

NOTE

For LAPACK, Intel MKL offers two types of the Fortran 95 interfaces:

- using `mkl_lapack.fi` only through the `include 'mkl_lapack.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
- using `lapack.f90` that includes improved interfaces. This file is used to generate the module files `lapack95.mod` and `f95_precision.mod`. See also the section "Fortran 95 interfaces and wrappers to LAPACK and BLAS" of the *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the LAPACK interface: `use lapack95` and `use f95_precision`.

The main conventions for the Fortran 95 interface are as follows:

- The names of arguments used in Fortran 95 call are typically the same as for the respective generic (FORTRAN 77) interface. In rare cases, formal argument names may be different. For instance, `select` instead of `selctg`.
- Input arguments such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

Another type of generic arguments that are skipped in the Fortran 95 interface are arguments that represent workspace arrays (such as `work`, `rwork`, and so on). The only exception are cases when workspace arrays return significant information on output.

NOTE

Internally, workspace arrays are allocated by the Fortran 95 interface wrapper, and are of optimal size for the best performance of the routine.

An argument can also be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.

- Some generic arguments are declared as optional in the Fortran 95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it meets one of the following conditions:
 - If an argument value is completely defined by the presence or absence of another argument in the calling sequence, it can be declared optional. The difference from the skipped argument in this case is that the optional argument can have some meaningful values that are distinct from the value reconstructed by default. For example, if some argument (like `jobz`) can take only two values and one of these values directly implies the use of another argument, then the value of `jobz` can be uniquely reconstructed from the actual presence or absence of this second argument, and `jobz` can be omitted.
 - If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
 - If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.

- Argument *info* is declared as optional in the Fortran 95 interface. If it is present in the calling sequence, the value assigned to *info* is interpreted as follows:
 - If this value is more than -1000, its meaning is the same as in the FORTRAN 77 routine.
 - If this value is equal to -1000, it means that there is not enough work memory.
 - If this value is equal to -1001, incompatible arguments are present in the calling sequence.
 - If this value is equal to *-i*, the *i*th parameter (counting parameters in the FORTRAN 77 interface, not the Fortran 95 interface) had an illegal value.
- Optional arguments are given in square brackets in the Fortran 95 call syntax.

The "Fortran 95 Notes" subsection at the end of the topic describing each routine details concrete rules for reconstructing the values of the omitted optional parameters.

Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation

The following list presents general digressions of the Intel MKL LAPACK95 implementation from the Netlib analog:

- The Intel MKL Fortran 95 interfaces are provided for pure procedures.
- Names of interfaces do not contain the `LA_` prefix.
- An optional array argument always has the `target` attribute.
- Functionality of the Intel MKL LAPACK95 wrapper is close to the FORTRAN 77 original implementation in the `getrf`, `gbtrf`, and `potrf` interfaces.
- If *jobz* argument value specifies presence or absence of *z* argument, then *z* is always declared as optional and *jobz* is restored depending on whether *z* is present or not.
- To avoid double error checking, processing of the *info* argument is limited to checking of the allocated memory and disarranging of optional arguments.
- If an argument that is present in the list of arguments completely defines another argument, the latter is always declared as optional.

You can transform an application that uses the Netlib LAPACK interfaces to ensure its work with the Intel MKL interfaces providing that:

- The application is correct, that is, unambiguous, compiler-independent, and contains no errors.
- Each routine name denotes only one specific routine. If any routine name in the application coincides with a name of the original Netlib routine (for example, after removing the `LA_` prefix) but denotes a routine different from the Netlib original routine, this name should be modified through context name replacement.

You should transform your application in the following cases:

- When using the Netlib routines that differ from the Intel MKL routines only by the `LA_` prefix or in the array attribute `target`. The only transformation required in this case is context name replacement.
- When using Netlib routines that differ from the Intel MKL routines by the `LA_` prefix, the `target` array attribute, and the names of formal arguments. In the case of positional passing of arguments, no additional transformation except context name replacement is required. In the case of the keywords passing of arguments, in addition to the context name replacement the names of mismatching keywords should also be modified.
- When using the Netlib routines that differ from the respective Intel MKL routines by the `LA_` prefix, the `target` array attribute, sequence of the arguments, arguments missing in Intel MKL but present in Netlib and, vice versa, present in Intel MKL but missing in Netlib. Remove the differences in the sequence and range of the arguments in process of all the transformations when you use the Netlib routines specified by this bullet and the preceding bullet.
- When using the `getrf`, `gbtrf`, and `potrf` interfaces, that is, new functionality implemented in Intel MKL but unavailable in the Netlib source. To override the differences, build the desired functionality explicitly with the Intel MKL means or create a new subroutine with the new functionality, using specific MKL interfaces corresponding to LAPACK 77 routines. You can call the LAPACK 77 routines directly but using the new Intel MKL interfaces is preferable. Note that if the transformed application calls `getrf`, `gbtrf` or `potrf` without controlling arguments *rcond* and *norm*, just context name replacement is enough in modifying the calls into the Intel MKL interfaces, as described in the first bullet above. The Netlib functionality is preserved in such cases.

- When using the Netlib auxiliary routines. In this case, call a corresponding subroutine directly, using the Intel MKL LAPACK 77 interfaces.

Transform your application as follows:

1. Make sure conditions a. and b. are met.
2. Select Netlib LAPACK 95 calls. For each call, do the following:
 - Select the type of digression and do the required transformations.
 - Revise results to eliminate unneeded code or data, which may appear after several identical calls.
3. Make sure the transformations are correct and complete.

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- *Full storage*: an m -by- n matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} ($i = 1..m, j = 1..n$), and stored in the array element $a(i, j)$.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: an m -by- n band matrix with kl sub-diagonals and ku superdiagonals is stored compactly in a two-dimensional array ab with $kl+ku+1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.
- *Rectangular Full Packed (RFP) storage*: the upper or lower triangle of the matrix is packed combining the full and packed storage schemes. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

Generally in LAPACK routines, arrays that hold matrices in packed storage have names ending in p ; arrays with matrices in band storage have names ending in b ; arrays with matrices in the RFP storage have names ending in fp .

For more information on matrix storage schemes, see "Matrix Arguments" in Appendix B.

Mathematical Notation

Descriptions of LAPACK routines use the following notation:

A^H	For an M -by- N matrix A , denotes the conjugate transposed N -by- M matrix with elements: $a_{ij}^H = \overline{a_{ji}}$ For a real-valued matrix, $A^H = A^T$.
$x \cdot y$	The <i>dot product</i> of two vectors, defined as: $x \cdot y = \sum_i x_i \overline{y_i}$
$Ax = b$	A system of linear equations with an n -by- n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.
$ x $	the vector with elements $ x_i $ (absolute values of x_i).

$ A $	the matrix with elements $ a_{ij} $ (absolute values of a_{ij}).
$\ x\ _\infty = \max_i x_i $	The <i>infinity-norm</i> of the vector x .
$\ A\ _\infty = \max_i \sum_j a_{ij} $	The <i>infinity-norm</i> of the matrix A .
$\ A\ _1 = \max_j \sum_i a_{ij} $	The <i>one-norm</i> of the matrix A . $\ A\ _1 = \ A^T\ _\infty = \ A^H\ _\infty$
$\ x\ _2$	The <i>2-norm</i> of the vector x : $\ x\ _2 = (\sum_i x_i ^2)^{1/2} = \ x\ _E$ (see the definition for <i>Euclidean norm</i> in this section).
$\ A\ _2$	The <i>2-norm</i> (or <i>spectral norm</i>) of the matrix A . $\ A\ _2 = \max_i \sigma_i, \ A\ _2^2 = \max_{\ x\ _E=1} (Ax \cdot Ax)$
$\ A\ _E$	The <i>Euclidean norm</i> of the matrix A : $\ A\ _E^2 = \sum_i \sum_j a_{ij} ^2$.
$\kappa(A) = \ A\ \cdot \ A^{-1}\ $	The <i>condition number</i> of the matrix A .
λ_j	<i>Eigenvalues</i> of the matrix A (for the definition of eigenvalues, see Eigenvalue Problems).
σ_i	<i>Singular values</i> of the matrix A . They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of A and b) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution x .

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)(x + \delta x) = (b + \delta b)$, then this estimate, given up to linear terms of perturbations, holds:

$$\frac{\| \delta x \|}{\| x \|} \leq c(n) \kappa(A) \varepsilon$$

where $A + \delta A$ is nonsingular and

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = \|A\| \|A^{-1}\|$ called the *condition number* of A .

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the *machine precision*, defined as the smallest positive number x such that $1 + x > 1$; and $c(n)$ is a modest function of the matrix order n . The corresponding solution error is

$$\| \delta x \| / \| x \| \leq c(n) \kappa(A) \varepsilon. \text{ (The value of } c(n) \text{ is seldom greater than } 10n.)$$

NOTE

Machine precision depends on the data type used.

Thus, if your matrix A is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x can also be large; you might even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see [Routines for Estimating the Condition Number](#)) and also give you a more precise estimate for the actual solution error (see [Refining the Solution and Estimating Its Error](#)).

Linear Equation Routines

This section describes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix (except for RFP matrices)
- solving a system of linear equations
- estimating the condition number of a matrix (except for RFP matrices)
- refining the solution of linear equations and computing its error bounds (except for RFP matrices)
- inverting the matrix.

To solve a particular problem, you can call two or more [computational routines](#) or call a corresponding [driver routine](#) that combines several tasks in one call. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

Computational Routines

Table "Computational Routines for Systems of Equations with Real Matrices" lists the LAPACK computational routines (FORTRAN 77 and Fortran 95 interfaces) for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. Table "Computational Routines for Systems of Equations with Complex Matrices" lists similar routines for *complex* matrices. Respective routine names in the Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<code>?getrf</code>	<code>?geequ,</code> <code>?geequb</code>	<code>?getrs</code>	<code>?gecon</code>	<code>?gerfs,</code> <code>?gerfsx</code>	<code>?getri</code>
general band	<code>?gbtrf</code>	<code>?gbequ,</code> <code>?gbequb</code>	<code>?gbtrs</code>	<code>?gbcon</code>	<code>?gbrfs,</code> <code>?gbrfsx</code>	
general tridiagonal	<code>?gttrf</code>		<code>?gttrs</code>	<code>?gtcon</code>	<code>?gtrfs</code>	
diagonally dominant tridiagonal	<code>?dttrfb</code>		<code>?dttrsb</code>			
symmetric positive-definite	<code>?potrf</code>	<code>?poequ,</code> <code>?poequb</code>	<code>?potrs</code>	<code>?pocon</code>	<code>?porfs,</code> <code>?porfsx</code>	<code>?potri</code>

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
symmetric positive-definite, RFP storage	?pftrf		?pftrs			?pftri
symmetric positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf	?syequb	?sytrs	?sycon	?syrfs,	?sytri
	? sytrf_ro ok		? sytrs_r ook	? sycon_roo k	?syrfssx	?sytri_rook ? sytri2 ? sytri2x
			?sytrs2			
symmetric indefinite, packed storage	?sptrf mkl_? spffrt2, mkl_? spffrtx		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? denotes s (single precision) or d (double precision) for the FORTRAN 77 interface.

Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ, ? geequb	?getrs	?gecon	?gerfs, ? gerfsx	?getri
general band	?gbtrf	?gbequ, ? gbequb	?gbtrs	?gbcon	?gbrfs, ? gbrfsx	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ, ? poequb	?potrs	?pocon	?porfs, ? porfsx	?potri

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, RFP storage	?pftrf		?pftrs			?pftri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf ? hetrf_ro ok	?heequb	?hetrs ? hetrs_r ook ? hetrs2	?hecon ? hecon_roo k	?herfs, ? herfsx	?hetri ? hetri_rook ? hetri2 ? hetri2x
symmetric indefinite	?sytrf ? sytrf_ro ok	?syequb	?sytrs ? sytrs_r ook ? sytrs2	?sycon ? sycon_roo k	?syrfs, ? syrfsex	?sytri ? sytri_rook ? sytri2 ? sytri2x
Hermitian indefinite, packed storage	?hptrf		?hptrs	?hpcon	?hprfs	?hptri
symmetric indefinite, packed storage	?sptrf mkl_ spffrt2, mkl_ spffrtx		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? stands for c (single precision complex) or z (double precision complex) for FORTRAN 77 interface.

Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization

- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of real symmetric positive-definite matrices with pivoting
- Cholesky factorization of Hermitian positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices with pivoting
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the LU factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, RFP, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

?getrf

Computes the LU factorization of a general m -by- n matrix.

Syntax

```
call sgetrf( m, n, a, lda, ipiv, info )
call dgetrf( m, n, a, lda, ipiv, info )
call cgetrf( m, n, a, lda, ipiv, info )
call zgetrf( m, n, a, lda, ipiv, info )
call getrf( a [,ipiv] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the LU factorization of a general m -by- n matrix A as

$$A = P^*L^*U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ; $n \geq 0$.
a	REAL for <code>sgetrf</code> DOUBLE PRECISION for <code>dgetrf</code> COMPLEX for <code>cgetrf</code> DOUBLE COMPLEX for <code>zgetrf</code> .

Array, size lda by $*$. Contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of array a .

Output Parameters

a Overwritten by L and U . The unit diagonal elements of L are not stored.

$ipiv$ INTEGER.

Array, size at least $\max(1, \min(m, n))$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row $ipiv(i)$.

$info$ INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `getrf` interface are as follows:

a Holds the matrix A of size (m, n) .

$ipiv$ Holds the vector of length $\min(m, n)$.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(\min(m, n)) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(2/3)n^3 \quad \text{If } m = n,$$

$$(1/3)n^2(3m-n) \quad \text{If } m > n,$$

$$(1/3)m^2(3n-m) \quad \text{If } m < n.$$

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

`?getrs` to solve $A^*X = B$ or $A^T X = B$ or $A^H X = B$

`?gecon` to estimate the condition number of A

`?getri` to compute the inverse of A .

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

mkl_?getrfnpi

Performs LU factorization (complete or incomplete) of a general matrix without pivoting.

Syntax

```
call mkl_sgetrfnpi (m, n, nfact, a, lda, info)
call mkl_dgetrfnpi (m, n, nfact, a, lda, info )
call mkl_cgetrfnpi (m, n, nfact, a, lda, info )
call mkl_zgetrfnpi (m, n, nfact, a, lda, info )
call mkl_getrfnpi ( a [, nfact] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the LU factorization of a general m -by- n matrix A without using pivoting. It supports incomplete factorization. The factorization has the form:

$$A = L*U,$$

where L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Incomplete factorization has the form:

$$A = L * U + \tilde{A}$$

where L is lower trapezoidal with unit diagonal elements, U is upper trapezoidal, and \tilde{A} is the unfactored part of matrix A . See the application notes section for further details.

NOTE

Use [?getrf](#) if it is possible that the matrix is not diagonal dominant.

Input Parameters

The data types are given for the Fortran interface.

m	INTEGER. The number of rows in matrix A ; $m \geq 0$.
n	INTEGER. The number of columns in matrix A ; $n \geq 0$.
$nfact$	INTEGER. The number of rows and columns to factor; $0 \leq nfact \leq \min(m, n)$. Note that if $nfact < \min(m, n)$, incomplete factorization is performed.
a	REAL for <code>mkl_sgetrfnpi</code> DOUBLE PRECISION for <code>mkl_dgetrfnpi</code> COMPLEX for <code>mkl_cgetrfnpi</code>

DOUBLE COMPLEX for `mkl_zgetrfnpi`

Array of size $(lda, *)$. Contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of array a . $lda \geq \max(1, m)$.

Output Parameters

a Overwritten by L and U . The unit diagonal elements of L are not stored. When incomplete factorization is specified by setting $nfact < \min(m, n)$, a

also contains the unfactored submatrix \tilde{A}_{22} . See the application notes section for further details.

$info$ INTEGER. If $info=0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = i$, u_{ij} is 0. The requested factorization has been completed, but U is exactly singular. Division by 0 will occur if factorization is completed and factor U is used for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getrf` interface are as follows:

a Holds the matrix A of size (m, n) .

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, with

$$|E| \leq c(\min(m, n))\epsilon |L| |U|$$

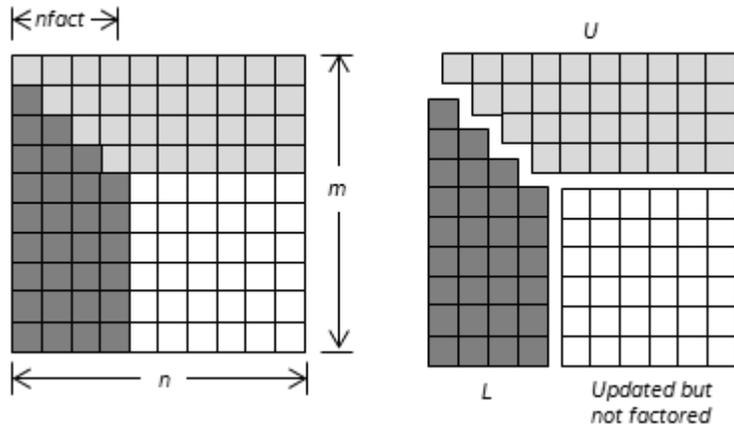
where $c(n)$ is a modest linear function of n , and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n = nfact$
$(1/3)n^2(3m-n)$	If $m > n = nfact$
$(1/3)m^2(3n-m)$	If $m = nfact < n$
$(2/3)n^3 - (n-nfact)^3$	If $m = n, nfact < \min(m, n)$
$(1/3)(n^2(3m-n) - (n-nfact)^2(3m - 2nfact - n))$	If $m > n > nfact$
$(1/3)(m^2(3n-m) - (m-nfact)^2(3n - 2nfact - m))$	If $nfact < m < n$

The number of operations for complex flavors is four times greater.

When incomplete factorization is specified, the first n_{fact} rows and columns are factored, with the update of the remaining rows and columns of A as follows:



If matrix A is represented as a block 2-by-2 matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

- A_{11} is a square matrix of order n_{fact} ,
- A_{21} is an $(m - n_{fact})$ -by- n_{fact} matrix,
- A_{12} is an n_{fact} -by- $(n - n_{fact})$ matrix, and
- A_{22} is an $(m - n_{fact})$ -by- $(n - n_{fact})$ matrix.

The result is

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \cdot \begin{bmatrix} U_1 & U_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix}$$

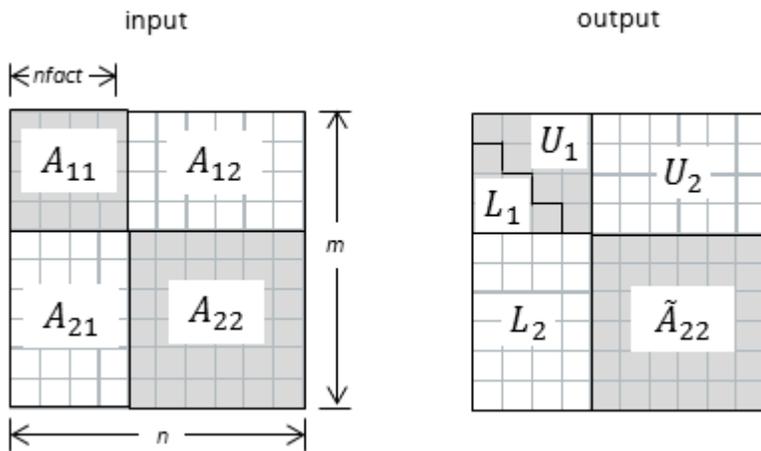
L_1 is a lower triangular square matrix of order n_{fact} with unit diagonal and U_1 is an upper triangular square matrix of order n_{fact} . L_1 and U_1 result from LU factorization of matrix A_{11} : $A_{11} = L_1 U_1$.

L_2 is an $(m - n_{fact})$ -by- n_{fact} matrix and $L_2 = A_{21} U_1^{-1}$. U_2 is an n_{fact} -by- $(n - n_{fact})$ matrix and $U_2 = L_1^{-1} A_{12}$.

\tilde{A}_{22} is an $(m - n_{fact})$ -by- $(n - n_{fact})$ matrix and $\tilde{A}_{22} = A_{22} - L_2 U_2$.

On exit, elements of the upper triangle U_1 are stored in place of the upper triangle of block A_{11} in array a ; elements of the lower triangle L_1 are stored in the lower triangle of block A_{11} in array a (unit diagonal

elements are not stored). Elements of L_2 replace elements of A_{21} ; U_2 replaces elements of A_{12} and replaces elements of A_{22} .



?gbtrf

Computes the LU factorization of a general m -by- n band matrix.

Syntax

```
call sgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call gbtrf( ab [,kl] [,m] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the LU factorization of a general m -by- n band matrix A with kl non-zero subdiagonals and ku non-zero superdiagonals, that is,

$$\bar{A} = P * L * U,$$

where P is a permutation matrix; L is lower triangular with unit diagonal elements and at most kl non-zero elements in each column; U is an upper triangular band matrix with $kl + ku$ superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional kl superdiagonals in U).

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m	INTEGER. The number of rows in matrix A ; $m \geq 0$.
n	INTEGER. The number of columns in matrix A ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.

ku INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab REAL for *sgbtrf*
 DOUBLE PRECISION for *dgbtrf*
 COMPLEX for *cgbtrf*
 DOUBLE COMPLEX for *zgbtrf*.
 Array, size *ldab* by *.
 The array *ab* contains the matrix *A* in band storage, in rows $kl + 1$ to $2*kl + ku + 1$; rows 1 to *kl* of the array need not be set. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:
 $ab(kl + ku + 1 + i - j, j) = a(i, j)$ for $\max(1, j - ku) \leq i \leq \min(m, j + kl)$.

ldab INTEGER. The leading dimension of the array *ab*. ($ldab \geq 2*kl + ku + 1$)

Output Parameters

ab Overwritten by *L* and *U*. *U* is stored as an upper triangular band matrix with $kl + ku$ superdiagonals in rows 1 to $kl + ku + 1$, and the multipliers used during the factorization are stored in rows $kl + ku + 2$ to $2*kl + ku + 1$.
 See Application Notes below for further details.

ipiv INTEGER.
 Array, size at least $\max(1, \min(m, n))$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row *i* was interchanged with row *ipiv*(*i*).

info If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *gbtrf* interface are as follows:

ab Holds the array *A* of size $(2*kl+ku+1, n)$.

ipiv Holds the vector of length $\min(m, n)$.

kl If omitted, assumed $kl = ku$.

ku Restored as $ku = lda - 2*kl - 1$.

m If omitted, assumed $m = n$.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(kl+ku+1) \varepsilon P \|L\| \|U\|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that kl and ku are much less than $\min(m, n)$.

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry	on exit
$\begin{bmatrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$	$\begin{bmatrix} * & * & * & U_{14} & U_{25} & U_{36} \\ * & * & U_{13} & U_{24} & U_{35} & U_{46} \\ * & U_{12} & U_{23} & U_{34} & U_{45} & U_{56} \\ U_{11} & U_{22} & U_{33} & U_{44} & U_{55} & U_{66} \\ M_{21} & M_{32} & M_{43} & M_{54} & M_{65} & * \\ M_{31} & M_{42} & M_{53} & M_{64} & * & * \end{bmatrix}$

Elements marked * are not used; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

After calling this routine with $m = n$, you can call the following routines:

`gbtrs` to solve $A * X = B$ or $A^T * X = B$ or $A^H * X = B$

`gbcon` to estimate the condition number of A .

See Also

[mkl_progress](#)

[Matrix Storage Schemes](#)

?gttrf

Computes the LU factorization of a tridiagonal matrix.

Syntax

`call sgtrf(n, dl, d, du, du2, ipiv, info)`

`call dgtrf(n, dl, d, du, du2, ipiv, info)`

`call cgtrf(n, dl, d, du, du2, ipiv, info)`

`call zgtrf(n, dl, d, du, du2, ipiv, info)`

`call gttrf(dl, d, du, du2 [, ipiv] [,info])`

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the LU factorization of a real or complex tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

$$A = L^*U,$$

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

Input Parameters

n INTEGER. The order of the matrix A ; $n \geq 0$.

dl, d, du REAL for `sgttrf`
 DOUBLE PRECISION for `dgttrf`
 COMPLEX for `cgttrf`
 DOUBLE COMPLEX for `zgttrf`.

Arrays containing elements of A .

The array dl of dimension $(n - 1)$ contains the subdiagonal elements of A .

The array d of dimension n contains the diagonal elements of A .

The array du of dimension $(n - 1)$ contains the superdiagonal elements of A .

Output Parameters

dl Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A . The matrix L has unit diagonal elements, and the $(n-1)$ elements of dl form the subdiagonal. All other elements of L are zero.

d Overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

du Overwritten by the $(n-1)$ elements of the first superdiagonal of U .

$du2$ REAL for `sgttrf`
 DOUBLE PRECISION for `dgttrf`
 COMPLEX for `cgttrf`
 DOUBLE COMPLEX for `zgttrf`.

Array, dimension $(n - 2)$. On exit, $du2$ contains $(n-2)$ elements of the second superdiagonal of U .

$ipiv$ INTEGER.

Array, dimension (n) . The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv(i)$. $ipiv(i)$ is always i or $i+1$; $ipiv(i) = i$ indicates a row interchange was not required.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gttrf` interface are as follows:

<i>dl</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length n .

Application Notes

<code>?gbtrs</code>	to solve $A * X = B$ or $A^T * X = B$ or $A^H * X = B$
<code>?gbcon</code>	to estimate the condition number of <i>A</i> .

?dttrfb

Computes the factorization of a diagonally dominant tridiagonal matrix.

Syntax

```
call sdttrfb( n, dl, d, du, info )
call ddttrfb( n, dl, d, du, info )
call cdttrfb( n, dl, d, du, info )
call zdttrfb( n, dl, d, du, info )
call dttrfb( dl, d, du [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The `?dttrfb` routine computes the factorization of a real or complex tridiagonal matrix *A* with the BABE (Burning At Both Ends) algorithm without pivoting. The factorization has the form

$$\bar{A} = L_1 * U * L_2$$

where

- L_1 and L_2 are unit lower bidiagonal with k and $n - k - 1$ subdiagonal elements, respectively, where $k = n/2$, and
- U is an upper bidiagonal matrix with nonzeros in only the main diagonal and first superdiagonal.

Input Parameters

n	INTEGER. The order of the matrix A ; $n \geq 0$.
dl, d, du	REAL for <code>sdttrfb</code> DOUBLE PRECISION for <code>ddttrfb</code> COMPLEX for <code>cdttrfb</code> DOUBLE COMPLEX for <code>zdttrfb</code> . Arrays containing elements of A . The array dl of dimension $(n - 1)$ contains the subdiagonal elements of A . The array d of dimension n contains the diagonal elements of A . The array du of dimension $(n - 1)$ contains the superdiagonal elements of A .

Output Parameters

dl	Overwritten by the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A .
d	Overwritten by the n diagonal element reciprocals of the upper triangular matrix U from the factorization of A .
du	Overwritten by the $(n-1)$ elements of the superdiagonal of U .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i :

$$1 < i < n, \text{ and } |d_1| > |du_1|, |d_n| > |dl_{n-1}|$$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems are free from the numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gttrf](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
 - Applying the `?dtrfb` factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.
-

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spotrf( uplo, n, a, lda, info )
call dpotrf( uplo, n, a, lda, info )
call cpotrf( uplo, n, a, lda, info )
call zpotrf( uplo, n, a, lda, info )
call potrf( a [, uplo] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A :

$$\begin{aligned}
 A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\
 A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L'
 \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored and how A is factored:</p> <p>If <code>uplo = 'U'</code>, the array <code>a</code> stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.</p> <p>If <code>uplo = 'L'</code>, the array <code>a</code> stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.</p>
<code>n</code>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>

<i>a</i>	REAL for <code>spotrf</code> DOUBLE PRECISION for <code>dpotrf</code> COMPLEX for <code>cpotrf</code> DOUBLE COMPLEX for <code>zpotrf</code> . Array, size $(lda, *)$. The array <i>a</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix <i>A</i> .

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `potrf` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?potrs</code>	to solve $A * X = B$
<code>?pocon</code>	to estimate the condition number of <i>A</i>

`?potri` to compute the inverse of A .

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?pstrf

Computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix.

Syntax

```
call spstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call zpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix. The form of the factorization is:

$$\begin{aligned}
 P^T * A * P &= U^T * U, \text{ if } uplo = 'U' \text{ for real flavors,} \\
 P^T * A * P &= U^H * U, \text{ if } uplo = 'U' \text{ for complex flavors,} \\
 P^T * A * P &= L * L^T, \text{ if } uplo = 'L' \text{ for real flavors,} \\
 P^T * A * P &= L * L^H, \text{ if } uplo = 'L' \text{ for complex flavors,}
 \end{aligned}$$

where P is a permutation matrix stored as vector `piv`, and U and L are upper and lower triangular matrices, respectively.

This algorithm does not attempt to check that A is positive semidefinite. This version of the algorithm calls level 3 BLAS.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the upper triangular part of the matrix A , and the strictly lower triangular part of the matrix is not referenced. If <code>uplo = 'L'</code> , the array <code>a</code> stores the lower triangular part of the matrix A , and the strictly upper triangular part of the matrix is not referenced.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>a, work</code>	REAL for <code>spstrf</code> DOUBLE PRECISION for <code>dpstrf</code>

COMPLEX for `cpstrf`

DOUBLE COMPLEX for `zpstrf`.

Array `a`, size $(lda, *)$. The array `a` contains either the upper or the lower triangular part of the matrix A (see `uplo`). The second dimension of `a` must be at least $\max(1, n)$.

`work(*)` is a workspace array. The dimension of `work` is at least $\max(1, 2*n)$.

`tol`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

User defined tolerance. If $tol < 0$, then $n*\epsilon*\max(A_{k,k})$, where ϵ is the machine precision, will be used (see [Error Analysis](#) for the definition of machine precision). The algorithm terminates at the $(k-1)$ -st step, if the pivot $\leq tol$.

`lda`

INTEGER. The leading dimension of `a`; at least $\max(1, n)$.

Output Parameters

`a`

If `info = 0`, the factor U or L from the Cholesky factorization is as described in *Description*.

`piv`

INTEGER.

Array, size at least $\max(1, n)$. The array `piv` is such that the nonzero entries are $P_{piv(k),k}$ ($1 \leq k \leq n$).

`rank`

INTEGER.

The rank of `a` given by the number of steps the algorithm completed.

`info`

INTEGER. If `info = 0`, the execution is successful.

If `info = -k`, the k -th argument had an illegal value.

If `info > 0`, the matrix A is either rank deficient with a computed rank as returned in `rank`, or is not positive semidefinite.

See Also

Matrix Storage Schemes

?pstrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format.

Syntax

```
call spstrf( transr, uplo, n, a, info )
```

```
call dpstrf( transr, uplo, n, a, info )
```

```
call cpstrf( transr, uplo, n, a, info )
```

```
call zpstrf( transr, uplo, n, a, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, a Hermitian positive-definite matrix A :

$$\begin{aligned}
 A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo='U' \\
 A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo='L'
 \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>transr</i>	<p>CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).</p> <p>If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP A is stored.</p> <p>If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP A is stored.</p> <p>If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP A is stored.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A.</p> <p>If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A; $n \geq 0$.</p>
<i>a</i>	<p>REAL for <code>spftrf</code></p> <p>DOUBLE PRECISION for <code>dpftrf</code></p> <p>COMPLEX for <code>cpftrf</code></p> <p>DOUBLE COMPLEX for <code>zpftrf</code>.</p> <p>Array, size $(n * (n + 1) / 2)$. The array <i>a</i> contains the matrix A in the RFP format.</p>

Output Parameters

<i>a</i>	<p><i>a</i> is overwritten by the Cholesky factor U or L, as specified by <i>uplo</i> and <i>trans</i>.</p>
<i>info</i>	<p>INTEGER. If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i> = $-i$, the i-th parameter had an illegal value.</p>

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

See Also

Matrix Storage Schemes

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

```
call spptrf( uplo, n, ap, info )
call dpptrf( uplo, n, ap, info )
call cpptrf( uplo, n, ap, info )
call zpptrf( uplo, n, ap, info )
call pptrf( ap [, uplo] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array ap , and how A is factored: If $uplo = 'U'$, the array ap stores the upper triangular part of the matrix A , and A is factored as $U^H * U$. If $uplo = 'L'$, the array ap stores the lower triangular part of the matrix A ; A is factored as $L * L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>ap</i>	REAL for spptrf DOUBLE PRECISION for dpptrf

COMPLEX for `cpptrf`

DOUBLE COMPLEX for `zpptrf`.

Array, size at least $\max(1, n(n+1)/2)$. The array `ap` contains either the upper or the lower triangular part of the matrix `A` (as specified by `uplo`) in packed storage (see [Matrix Storage Schemes](#)).

Output Parameters

`ap`

Overwritten by the Cholesky factor `U` or `L`, as specified by `uplo`.

`info`

INTEGER. If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, the leading minor of order *i* (and therefore the matrix `A` itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix `A`.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pptrf` interface are as follows:

`ap`

Holds the array `A` of size $(n*(n+1)/2)$.

`uplo`

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo = 'U'`, the computed factor `U` is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\epsilon |U^H| |U|, |e_{ij}| \leq c(n)\epsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

A similar estimate holds for `uplo = 'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?pptrs`

to solve $A*X = B$

`?ppcon`

to estimate the condition number of `A`

`?pptri`

to compute the inverse of `A`.

See Also

[mkl_progress](#)

[Matrix Storage Schemes](#)

?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbtrf( uplo, n, kd, ab, ldab, info )
call dpbtrf( uplo, n, kd, ab, ldab, info )
call cpbtrf( uplo, n, kd, ab, ldab, info )
call zpbtrf( uplo, n, kd, ab, ldab, info )
call pbtrf( ab [, uplo] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix A :

$$\begin{array}{ll}
 A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\
 A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L'
 \end{array}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array ab , and how A is factored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>ab</i>	REAL for spbtrf DOUBLE PRECISION for dpbtrf COMPLEX for cpbtrf DOUBLE COMPLEX for zpbtrf. Array, size $(ldab, *)$. The array ab contains either the upper or the lower triangular part of the matrix A (as specified by $uplo$) in band storage (see Matrix Storage Schemes). The second dimension of ab must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*. ($ldab \geq kd + 1$)

Output Parameters

ab The upper or lower triangular part of *A* (in band storage) is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info INTEGER. If *info*=0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbtrf` interface are as follows:

ab Holds the array *A* of size $(kd+1, n)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(kd + 1)\epsilon |U^H| |U|, |e_{ij}| \leq c(kd + 1)\epsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of *n*, and ϵ is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kd* is much less than *n*.

After calling this routine, you can call the following routines:

`?pbtrs` to solve $A * X = B$

`?pbcon` to estimate the condition number of *A*.

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?pttrf

Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

call `spttrf(n, d, e, info)`

```

call dpttrf( n, d, e, info )
call cpttrf( n, d, e, info )
call zpttrf( n, d, e, info )
call pttrf( d, e [,info] )

```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix A :

$A = L*D*L^T$ for real flavors, or

$A = L*D*L^H$ for complex flavors,

where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^T*D*U$ for real flavors, or $A = U^H*D*U$ for complex flavors, where U is unit upper bidiagonal.

Input Parameters

n INTEGER. The order of the matrix A ; $n \geq 0$.

d REAL for `spttrf`, `cpttrf`
 DOUBLE PRECISION for `dpttrf`, `zpttrf`.
 Array, dimension (n). Contains the diagonal elements of A .

e REAL for `spttrf`
 DOUBLE PRECISION for `dpttrf`
 COMPLEX for `cpttrf`
 DOUBLE COMPLEX for `zpttrf`.
 Array, dimension ($n - 1$). Contains the subdiagonal elements of A .

Output Parameters

d Overwritten by the n diagonal elements of the diagonal matrix D from the $L*D*L^T$ (for real flavors) or $L*D*L^H$ (for complex flavors) factorization of A .

e Overwritten by the ($n - 1$) sub-diagonal elements of the unit bidiagonal factor L or U from the factorization of A .

$info$ INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d(n) \leq 0$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pstrf` interface are as follows:

<code>d</code>	Holds the vector of length n .
<code>e</code>	Holds the vector of length $(n-1)$.

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
call ssytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call sytrf( a [, uplo] [, ipiv] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

if $uplo='U'$, $A = U*D*U^T$
 if $uplo='L'$, $A = L*D*L^T$,

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Routines](#) section for details.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored and how A is factored:
	If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and A is factored as $U*D*U^T$.

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

a REAL for `ssytrf`
 DOUBLE PRECISION for `dsytrf`
 COMPLEX for `csytrf`
 DOUBLE COMPLEX for `zsytrf`.

Array, size $(lda, *)$. The array a contains either the upper or the lower triangular part of the matrix A (see $uplo$). The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

$work$ Same type as a . A workspace array, dimension at least $\max(1, lwork)$.

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See [Application Notes](#) for the suggested value of $lwork$.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

$work(1)$ If $info=0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$ipiv$ INTEGER.
 Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.
 If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.
 If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$info$ INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, D_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrf` interface are as follows:

<code>a</code>	holds the matrix A of size (n, n)
<code>ipiv</code>	holds the vector of length n
<code>uplo</code>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a .

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?sytrs</code>	to solve $A * X = B$
<code>?sycon</code>	to estimate the condition number of A
<code>?sytri</code>	to compute the inverse of A .

If $uplo = 'U'$, then $A = U^*D^*U'$, where

$$U = P(n)*U(n)* \dots *P(k)*U(k)*\dots,$$

that is, U is a product of terms $P(k)*U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$

$k-s \quad s \quad n-k$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If $uplo = 'L'$, then $A = L^*D^*L'$, where

$$L = P(1)*L(1)* \dots *P(k)*L(k)*\dots,$$

that is, L is a product of terms $P(k)*L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$

$k-1 \quad s \quad n-k-s+1$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

[Matrix Storage Schemes](#)

?sytrf_rook

Computes the bounded Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
call ssytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call sytrf_rook( a [, uplo] [, ipiv] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a real/complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The form of the factorization is:

if $uplo='U'$, $A = U*D*U^T$
 if $uplo='L'$, $A = L*D*L^T$,

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and A is factored as $U*D*U^T$. If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and A is factored as $L*D*L^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for ssytrf_rook DOUBLE PRECISION for dsytrf_rook COMPLEX for csytrf_rook DOUBLE COMPLEX for zsytrf_rook. Array, size ($lda, *$). The array a contains either the upper or the lower triangular part of the matrix A (see $uplo$). The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.

<i>work</i>	Same type as <i>a</i> . A workspace array, dimension at least $\max(1, lwork)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See ?sytrf Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>).
<i>work</i> (1)	If $info=0$, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> . If $ipiv(k) > 0$, then rows and columns <i>k</i> and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block. If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block. If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$, D_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular. Division by 0 will occur if you use <i>D</i> for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrf_rook` interface are as follows:

<i>a</i>	holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>)
<i>ipiv</i>	holds the vector of length <i>n</i>

`uplo` must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

- `?sytrs_rook` to solve $A*X = B$
- `?sycon_rook` to estimate the condition number of A
- `?sytri_rook` to compute the inverse of A .

If `uplo = 'U'`, then $A = U*D*U'$, where

$$U = P(n)*U(n)* \dots *P(k)*U(k)* \dots,$$

that is, U is a product of terms $P(k)*U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv(k)`.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If `uplo = 'L'`, then $A = L*D*L'$, where

$$L = P(1)*L(1)* \dots *P(k)*L(k)* \dots,$$

that is, L is a product of terms $P(k)*L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv(k)`.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

Matrix Storage Schemes

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
call chetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

```
call zhetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

```
call hetrf( a [, uplo] [, ipiv] [, info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U', A = U*D*UH
if uplo='L', A = L*D*LH,
```

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Routines](#) section for details.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$.

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.

n INTEGER. The order of matrix A ; $n \geq 0$.

$a, work$ COMPLEX for `chetrf`
 DOUBLE COMPLEX for `zhetrf`.
 Arrays, size $(lda, *)$, $work(*)$.

The array a contains the upper or the lower triangular part of the matrix A (see $uplo$). The second dimension of a must be at least $\max(1, n)$.

$work(*)$ is a workspace array of dimension at least $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See [Application Notes](#) for the suggested value of $lwork$.

Output Parameters

a The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$ipiv$ INTEGER.
 Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$info$ INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrf` interface are as follows:

<code>a</code>	holds the matrix A of size (n, n)
<code>ipiv</code>	holds the vector of length n
<code>uplo</code>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If A is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in D .

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a .

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hetrs</code>	to solve $A * X = B$
<code>?hecon</code>	to estimate the condition number of A
<code>?hetri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?hetrf_rook

Computes the bounded Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
call chetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call hetrf_rook( a [, uplo] [, ipiv] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman diagonal pivoting method:

```
if uplo='U',  $A = U^*D*U^H$ 
if uplo='L',  $A = L^*D*L^H$ ,
```

where A is the input matrix, U (or L) is a product of permutation and unit upper (or lower) triangular matrices, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for chetrf_rook COMPLEX*16 for zhetrf_rook. Arrays, size (<i>lda</i> , *), <i>work</i> (*). The array <i>a</i> contains the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <i>a</i> is not referenced. <i>work</i> (*) is a workspace array of dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).
 The length of *work*. $lwork \geq 1$. For best performance $lwork \geq n * nb$, where *nb* is the block size returned by [ilaenv](#).
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

Output Parameters

a The block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* (see Application Notes for further details).

work(1) If $info = 0$, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv INTEGER.
 Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of *D*.

- If $uplo = 'U'$:
 If $ipiv(k) > 0$, then rows and columns *k* and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.
 If $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns *k* and $-ipiv(k)$ were interchanged and rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, $D_{k-1:k,k-1:k}$ is a 2-by-2 diagonal block.
- If $uplo = 'L'$:
 If $ipiv(k) > 0$, then rows and columns *k* and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.
 If $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns *k* and $-ipiv(k)$ were interchanged and rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, $D_{k:k+1,k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the *i*-th parameter had an illegal value.
 If $info = i$, D_{ii} is exactly 0. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by 0 will occur if you use *D* for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrf_rook` interface are as follows:

<code>a</code>	holds the matrix A of size (n, n)
<code>ipiv</code>	holds the vector of length n
<code>uplo</code>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

If `uplo = 'U'`, then $A = U^*D^*U^H$, where

$$U = P(n)*U(n)* \dots *P(k)U(k)* \dots,$$

i.e., U is a product of terms $P(k)*U(k)$, where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by `ipiv(k)`, and $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$\begin{pmatrix} I & v & 0 \\ 0 & 0 & I \end{pmatrix}_{k-s} U(k) = \begin{pmatrix} 0 & I & 0 \\ & & k-s \end{pmatrix}_{n-k} \begin{pmatrix} & & 0 \\ & & I \end{pmatrix}_{k-ssn-k}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$, and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If `uplo = 'L'`, then $A = L^*D^*L^H$, where

$$L = P(1)*L(1)* \dots *P(k)*L(k)* \dots,$$

i.e., L is a product of terms $P(k)*L(k)$, where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by `ipiv(k)`, and $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & vI & \end{pmatrix}_{k-1} \begin{pmatrix} & & 0 \\ & & I \end{pmatrix}_{n-k-s+1} \begin{pmatrix} & & 0 \\ & & I \end{pmatrix}_{k-1} \begin{pmatrix} & & 0 \\ & & I \end{pmatrix}_{sn-k-s+1}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

[Matrix Storage Schemes](#)

?sptf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

```
call ssptf( uplo, n, ap, ipiv, info )
call dsptf( uplo, n, ap, ipiv, info )
call csptf( uplo, n, ap, ipiv, info )
call zsptf( uplo, n, ap, ipiv, info )
call sptf( ap [,uplo] [,ipiv] [,info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes the factorization of a real/complex symmetric matrix A stored in the packed format using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

$$\begin{aligned} \text{if } uplo='U', A &= U^*D^*U^T \\ \text{if } uplo='L', A &= L^*D^*L^T, \end{aligned}$$

where U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> and how A is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A, and A is factored as $U^*D^*U^T$.</p> <p>If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A, and A is factored as $L^*D^*L^T$.</p>
<i>n</i>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>
<i>ap</i>	<p>REAL for <code>ssptrf</code></p> <p>DOUBLE PRECISION for <code>dsptrf</code></p> <p>COMPLEX for <code>csptrf</code></p> <p>DOUBLE COMPLEX for <code>zsptrf</code>.</p> <p>Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).</p>

Output Parameters

<i>ap</i>	<p>The upper or lower triangle of A (as specified by <i>uplo</i>) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If <i>ipiv</i>(i) = $k > 0$, then $d_{i,i}$ is a 1-by-1 block, and the i-th row and column of A was interchanged with the k-th row and column.</p>

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spturf` interface are as follows:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L overwrite elements of the corresponding columns of the array *ap*, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in packed form.

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\epsilon P|U||D||U^T|P^T$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision. A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?spturs</code>	to solve $A*X = B$
<code>?spcon</code>	to estimate the condition number of A
<code>?sptri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?hptrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

```
call chptrf( uplo, n, ap, ipiv, info )
call zhptrf( uplo, n, ap, ipiv, info )
call hptrf( ap [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U',  $A = U^*D*U^H$ 
if uplo='L',  $A = L^*D*L^H$ ,
```

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is packed and how A is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A, and A is factored as U^*D*U^H.</p> <p>If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A, and A is factored as L^*D*L^H.</p>
<i>n</i>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>
<i>ap</i>	<p>COMPLEX for chptrf</p> <p>DOUBLE COMPLEX for zhptrf.</p> <p>Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).</p>

Output Parameters

<i>ap</i>	The upper or lower triangle of A (as specified by <i>uplo</i>) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D. If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i-th row and column of A was interchanged with the k-th row and column.</p> <p>If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$-th row and column of A was interchanged with the m-th row and column.</p> <p>If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$-th row and column of A was interchanged with the m-th row and column.</p>
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hptrf` interface are as follows:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the array *ap*, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array *a*.

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hptrs</code>	to solve $A * X = B$
<code>?hpcon</code>	to estimate the condition number of A
<code>?hptri</code>	to compute the inverse of A .

See Also

`mkl_progress`

Matrix Storage Schemes

`mkl_?spffrt2`, `mkl_?spffrtx`

Computes the partial LDL^T factorization of a symmetric matrix using packed storage.

Syntax

```
call mkl_sspffrt2( ap, n, ncolm, work, work2 )
call mkl_dspffrt2( ap, n, ncolm, work, work2 )
call mkl_cspffrt2( ap, n, ncolm, work, work2 )
call mkl_zspffrt2( ap, n, ncolm, work, work2 )
call mkl_sspffrtx( ap, n, ncolm, work, work2 )
call mkl_dspffrtx( ap, n, ncolm, work, work2 )
call mkl_cspffrtx( ap, n, ncolm, work, work2 )
call mkl_zspffrtx( ap, n, ncolm, work, work2 )
```

Include Files

- `mkl.fi`

Description

The routine computes the partial factorization $A = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix.

CAUTION

The routine assumes that the matrix A is factorizable. The routine does not perform pivoting and does not handle diagonal elements which are zero, which cause the routine to produce incorrect results without any indication.

Consider the matrix $A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix}$, where a is the element in the first row and first column of A , b is a column vector of size $n - 1$ containing the elements from the second through n -th column of A , C is the lower-right square submatrix of A , and I is the identity matrix.

The `mkl_?spffrt2` routine performs $ncolm$ successive factorizations of the form

$$A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix} = \begin{pmatrix} a & 0 \\ b & I \end{pmatrix} \begin{pmatrix} a^{-1} & 0 \\ 0 & C - ba^{-1}b^T \end{pmatrix} \begin{pmatrix} a & b^T \\ 0 & I \end{pmatrix}.$$

The `mkl_?spffrtx` routine performs $ncolm$ successive factorizations of the form

$$A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ ba^{-1} & I \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & C - ba^{-1}b^T \end{pmatrix} \begin{pmatrix} 1 & (ba^{-1})^T \\ 0 & I \end{pmatrix}.$$

The approximate number of floating point operations performed by real flavors of these routines is $(1/6)*ncolm*(2*ncolm^2 - 6*ncolm*n + 3*ncolm + 6*n^2 - 6*n + 7)$.

The approximate number of floating point operations performed by complex flavors of these routines is $(1/3)*ncolm*(4*ncolm^2 - 12*ncolm*n + 9*ncolm + 12*n^2 - 18*n + 8)$.

Input Parameters

<i>ap</i>	REAL for mkl_sspffrt2 and mkl_sspffrtx DOUBLE PRECISION for mkl_dspffrt2 and mkl_dspffrtx COMPLEX for mkl_cspffrt2 and mkl_cspffrtx DOUBLE COMPLEX for mkl_zspffrt2 and mkl_zspffrtx. Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the lower triangular part of the matrix <i>A</i> in packed storage (see Matrix Storage Schemes for <i>uplo</i> = 'L').
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>ncolm</i>	INTEGER. The number of columns to factor, $ncolm \leq n$.
<i>work, work2</i>	REAL for mkl_sspffrt2 and mkl_sspffrtx DOUBLE PRECISION for mkl_dspffrt2 and mkl_dspffrtx COMPLEX for mkl_cspffrt2 and mkl_cspffrtx DOUBLE COMPLEX for mkl_zspffrt2 and mkl_zspffrtx. Workspace arrays, size of each at least <i>n</i> .

Output Parameters

<i>ap</i>	Overwritten by the factor <i>L</i> . The first <i>ncolm</i> diagonal elements of the input matrix <i>A</i> are replaced with the diagonal elements of <i>D</i> . The subdiagonal elements of the first <i>ncolm</i> columns are replaced with the corresponding elements of <i>L</i> . The rest of the input array is updated as indicated in the Description section.
-----------	--

NOTE

Specifying $ncolm = n$ results in complete factorization $A = LDL^T$.

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

sgetrs dgetrs cgetrs zgetrs Reference

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

Syntax

```
call sgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call dgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call cgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call zgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call getrs( a, ipiv, b [, trans] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the following systems of linear equations:

$A * X = B$ if $trans = 'N'$,
 $A^T * X = B$ if $trans = 'T'$,
 $A^H * X = B$ if $trans = 'C'$ (for complex matrices only).

Before calling this routine, you must call `?getrf` to compute the LU factorization of A .

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations:
 If $trans = 'N'$, then $A * X = B$ is solved for X .
 If $trans = 'T'$, then $A^T * X = B$ is solved for X .
 If $trans = 'C'$, then $A^H * X = B$ is solved for X .

n INTEGER. The order of A ; the number of rows in B ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, b REAL for sgetrs
 DOUBLE PRECISION for dgetrs
 COMPLEX for cgetrs
 DOUBLE COMPLEX for zgetrs.
 Arrays: a (size lda by $*$), b (size ldb by $*$).
 The array a contains LU factorization of matrix A resulting from the call of `?getrf`.
 The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of a must be at least $\max(1, n)$ and the second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

ipiv INTEGER.
Array, size at least $\max(1, n)$. The *ipiv* array, as returned by [?getrf](#).

Output Parameters

b Overwritten by the solution matrix X .

info INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `getrs` interface are as follows:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size $(n, nrhs)$.

ipiv Holds the vector of length n .

trans Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$\|E\| \leq c(n) \varepsilon P \|L\| \|U\|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{matrix} | & | & \dots & | & \dots & | & | \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ & & & & & & | & | \end{matrix}$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\| \|A\| \|x\|}{\|x\|} \leq \|A^{-1}\| \|A\| = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?gecon`.

To refine the solution and estimate the error, call `?gerfs`.

See Also

Matrix Storage Schemes

`?gbtrs`

Solves a system of linear equations with an LU-factored band coefficient matrix, with multiple right-hand sides.

Syntax

```
call sgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call gbtrs( ab, b, ipiv, [, kl] [, trans] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the following systems of linear equations:

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Here A is an LU-factored general band matrix of order n with kl non-zero subdiagonals and ku nonzero superdiagonals. Before calling this routine, call `?gbtrf` to compute the LU factorization of A .

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
<i>n</i>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for <code>sgbtrs</code> DOUBLE PRECISION for <code>dgbtrs</code> COMPLEX for <code>cgbtrs</code> DOUBLE COMPLEX for <code>zgbtrs</code> .

Arrays: $ab(ldab, *)$, $b(l db, *)$.

The array ab contains elements of the LU factors of the matrix A as returned by [gbtrf](#). The second dimension of ab must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b at least $\max(1, nrhs)$.

<i>ldab</i>	INTEGER. The leading dimension of the array ab ; $ldab \geq 2 * kl + ku + 1$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gbtrf .

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbtrs` interface are as follows:

<i>ab</i>	Holds the array A of size $(2 * kl + ku + 1, n)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length $\min(m, n)$.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $lda - 2 * kl - 1$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1) \epsilon P |L| |U|$$

$c(k)$ is a modest linear function of k , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:



where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_{\infty} / \|x\|_{\infty} \leq \|A^{-1}\|_{\infty} \|A\|_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than $\min(m, n)$.

To estimate the condition number $\kappa_{\infty}(A)$, call `?gbcon`.

To refine the solution and estimate the error, call `?gbrfs`.

See Also

Matrix Storage Schemes

?gttrs

Solves a system of linear equations with a tridiagonal coefficient matrix using the LU factorization computed by ?gttrf.

Syntax

```
call sgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call dgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call cgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call zgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call gttrs( dl, d, du, du2, b, ipiv [, trans] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the following systems of linear equations with multiple right hand sides:

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Before calling this routine, you must call `?gttrf` to compute the LU factorization of A .

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If $trans = 'N'$, then $A * X = B$ is solved for X .

If $trans = 'T'$, then $A^T * X = B$ is solved for X .

If $trans = 'C'$, then $A^H * X = B$ is solved for X .

n	INTEGER. The order of A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.
$dl, d, du, du2, b$	<p>REAL for <code>sgttrs</code></p> <p>DOUBLE PRECISION for <code>dgttrs</code></p> <p>COMPLEX for <code>cgttrs</code></p> <p>DOUBLE COMPLEX for <code>zgttrs</code>.</p> <p>Arrays: $dl(n-1), d(n), du(n-1), du2(n-2), b(ldb, nrhs)$.</p> <p>The array dl contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A.</p> <p>The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.</p> <p>The array du contains the $(n-1)$ elements of the first superdiagonal of U.</p> <p>The array $du2$ contains the $(n-2)$ elements of the second superdiagonal of U.</p> <p>The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.</p>
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
$ipiv$	INTEGER. Array, size (n) . The $ipiv$ array, as returned by <code>?gttrf</code> .

Output Parameters

b	Overwritten by the solution matrix X .
$info$	<p>INTEGER. If $info=0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gttrs` interface are as follows:

dl	Holds the vector of length $(n-1)$.
d	Holds the vector of length n .
du	Holds the vector of length $(n-1)$.
$du2$	Holds the vector of length $(n-2)$.

<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x_0\|_\infty} \leq \text{cond}(A, x) \frac{\|E\|_\infty}{\|A\|_\infty}$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $7n$ (including *n* divisions) for real flavors and $34n$ (including $2n$ divisions) for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?gtcon](#).

To refine the solution and estimate the error, call [?gtrfs](#).

See Also

Matrix Storage Schemes

[?dttrs](#)

Solves a system of linear equations with a diagonally dominant tridiagonal coefficient matrix using the LU factorization computed by [?dttrfb](#).

Syntax

```
call sdttrs( trans, n, nrhs, dl, d, du, b, ldb, info )
call ddttrs( trans, n, nrhs, dl, d, du, b, ldb, info )
call cdttrs( trans, n, nrhs, dl, d, du, b, ldb, info )
call zdttrs( trans, n, nrhs, dl, d, du, b, ldb, info )
call dttrs( dl, d, du, b [, trans] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The [?dttrs](#) routine solves the following systems of linear equations with multiple right hand sides for *X*:

$A^*X = B$ if `trans='N'`,
 $A^T*X = B$ if `trans='T'`,
 $A^H*X = B$ if `trans='C'` (for complex matrices only).

Before calling this routine, call `?dtttrfb` to compute the factorization of A .

Input Parameters

`trans` CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations solved for X :
 If `trans = 'N'`, then $A^*X = B$.
 If `trans = 'T'`, then $A^T*X = B$.
 If `trans = 'C'`, then $A^H*X = B$.

`n` INTEGER. The order of A ; $n \geq 0$.

`nrhs` INTEGER. The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.

`dl, d, du, b` REAL for `sdttrsb`
 DOUBLE PRECISION for `ddttrsb`
 COMPLEX for `cdttrsb`
 DOUBLE COMPLEX for `zdttrsb`.
 Arrays: `dl(n-1)`, `d(n)`, `du(n-1)`, `b(ldb, nrhs)`.
 The array `dl` contains the $(n-1)$ multipliers that define the matrices L_1, L_2 from the factorization of A .
 The array `d` contains the n diagonal elements of the upper triangular matrix U from the factorization of A .
 The array `du` contains the $(n-1)$ elements of the superdiagonal of U .
 The array `b` contains the matrix B whose columns are the right-hand sides for the systems of equations.

`ldb` INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

`b` Overwritten by the solution matrix X .

`info` INTEGER. If `info = 0`, the execution is successful.
 If `info = -i`, the i -th parameter had an illegal value.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
call spotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call potrs( a, b [,uplo] [, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the system of linear equations $A * X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$$\begin{array}{ll}
 A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\
 A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L'
 \end{array}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [potrf](#) to compute the Cholesky factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', U is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data. If <i>uplo</i> = 'L', L is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<i>a, b</i>	REAL for spotrs DOUBLE PRECISION for dpotrs COMPLEX for cpotrs DOUBLE COMPLEX for zpotrs. Arrays: $a(lda, *)$, $b(ldb, *)$. The array a contains the factor U or L (see <i>uplo</i>) as returned by potrf . The second dimension of a must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix *X*.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `potrs` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *nrhs*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side *b* is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|$$

c(*n*) is a modest linear function of *n*, and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'. If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\|x - x_0\|_\infty \leq \text{cond}(A, x) \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty = \kappa_\infty(A) \|x\|_\infty$$

where $\text{cond}(A, x) = \|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector *b* is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pocon`.

To refine the solution and estimate the error, call `?porfs`.

See Also

Matrix Storage Schemes

?pftsr

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix using the Rectangular Full Packed (RFP) format.

Syntax

```
call spftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call dpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call cpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call zpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves a system of linear equations $A^*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A using the Cholesky factorization of A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

Before calling `?pftrs`, you must call `?pfftrf` to compute the Cholesky factorization of A . L stands for a lower triangular matrix and U for an upper triangular matrix.

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>transr</i>	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the untransposed factor of A is stored in RFP format. If <i>transr</i> = 'T', the transposed factor of A is stored in RFP format. If <i>transr</i> = 'C', the conjugate-transposed factor of A is stored in RFP format.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', U is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data. If <i>uplo</i> = 'L', L is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>a, b</i>	REAL for <code>spftrs</code> DOUBLE PRECISION for <code>dpftrs</code> COMPLEX for <code>cpftrs</code> DOUBLE COMPLEX for <code>zpftrs</code> . Arrays: $a(n * (n + 1) / 2)$, $b(ldb, nrhs)$.

The array *a* contains, in the RFP format, the factor *U* or *L* obtained by factorization of matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b The solution matrix *X*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

Matrix Storage Schemes

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
call spptrs( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs( uplo, n, nrhs, ap, b, ldb, info )
call pptrs( ap, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the system of linear equations $A * X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*, given the Cholesky factorization of *A*:

$$\begin{array}{ll}
 A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\
 A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L'
 \end{array}$$

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization of *A*.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix *A* has been factored:
If *uplo* = 'U', *U* is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data.

If $uplo = 'L'$, L is stored, where $A = L^*L^T$ for real data, $A = L^*L^H$ for complex data

n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
ap, b	REAL for <code>spptrs</code> DOUBLE PRECISION for <code>dpptrs</code> COMPLEX for <code>cpptrs</code> DOUBLE COMPLEX for <code>zpptrs</code> . Arrays: $ap(*), b(l\delta b, *)$ The size of ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by $uplo$, in <i>packed storage</i> (see Matrix Storage Schemes). The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pptrs` interface are as follows:

ap	Holds the array A of size $(n*(n+1)/2)$.
b	Holds the matrix B of size $(n, nrhs)$.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

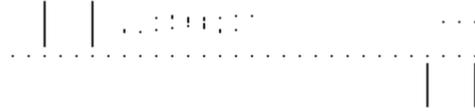
If $uplo = 'U'$, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$.

If x_0 is the true solution, the computed solution x satisfies this error bound:



where $\text{cond}(A, x) = \frac{\| |A^{-1}| |A| |x| \|_{\infty}}{\|x\|_{\infty}} \leq \|A^{-1}\|_{\infty} \|A\|_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?ppcon`.

To refine the solution and estimate the error, call `?pprfs`.

See Also
[Matrix Storage Schemes](#)

?pbtrs
Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band coefficient matrix.

Syntax

```
call spbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call pbtrs( ab, b [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for real data a system of linear equations $A * X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite *band* matrix A , given the Cholesky factorization of A :

$$\begin{array}{ll}
 A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\
 A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L'
 \end{array}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call `?pbtrf` to compute the Cholesky factorization of A in the band storage form.

Input Parameters

`uplo` CHARACTER*1. Must be 'U' or 'L'.
 Indicates how the input matrix A has been factored:

If $uplo = 'U'$, U is stored in ab , where $A = U^T * U$ for real matrices and $A = U^H * U$ for complex matrices.

If $uplo = 'L'$, L is stored in ab , where $A = L * L^T$ for real matrices and $A = L * L^H$ for complex matrices.

n	INTEGER. The order of matrix A ; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
ab, b	REAL for <code>spbtrs</code> DOUBLE PRECISION for <code>dpbtrs</code> COMPLEX for <code>cpbtrs</code> DOUBLE COMPLEX for <code>zpbtrs</code> . Arrays: $ab(ldab, *)$, $b(l db, *)$. The array ab contains the Cholesky factor, as returned by the factorization routine, in <i>band storage</i> form. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of ab must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.
$ldab$	INTEGER. The leading dimension of the array ab ; $ldab \geq kd + 1$.
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbtrs` interface are as follows:

ab	Holds the array A of size $(kd+1, n)$.
b	Holds the matrix B of size $(n, nrhs)$.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kd + 1)\varepsilon P |U^H| |U| \text{ or } |E| \leq c(kd + 1)\varepsilon P |L^H| |L|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?pbcon](#).

To refine the solution and estimate the error, call [?pbrfs](#).

See Also

Matrix Storage Schemes

[?pttrs](#)

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix using the factorization computed by [?pttrf](#).

Syntax

```
call spttrs( n, nrhs, d, e, b, ldb, info )
call dpttrs( n, nrhs, d, e, b, ldb, info )
call cpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call zpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call pttrs( d, e, b [,info] )
call pttrs( d, e, b [,uplo] [,info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine solves for X a system of linear equations $A*X = B$ with a symmetric (Hermitian) positive-definite tridiagonal matrix A . Before calling this routine, call [?pttrf](#) to compute the $L*D*L^T$ or U^T*D*U for real data and the $L*D*L^H$ or U^H*D*U factorization of A for complex data.

Input Parameters

`uplo` CHARACTER*1. Used for `cpttrs/zpttrs` only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored:

If $uplo = 'U'$, the array e stores the conjugated values of the superdiagonal of U , and A is factored as $U^H * D * U$.

If $uplo = 'L'$, the array e stores the subdiagonal of L , and A is factored as $L * D * L^H$.

n	INTEGER. The order of A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
d	REAL for <code>spttrs</code> , <code>cpttrs</code> DOUBLE PRECISION for <code>dpttrs</code> , <code>zpttrs</code> . Array, dimension (n). Contains the diagonal elements of the diagonal matrix D from the factorization computed by <code>?pttrf</code> .
e, b	REAL for <code>spttrs</code> DOUBLE PRECISION for <code>dpttrs</code> COMPLEX for <code>cpttrs</code> DOUBLE COMPLEX for <code>zpttrs</code> . Arrays: $e(n-1)$, $b(ldb, nrhs)$. The array e contains the $(n-1)$ sub-diagonal elements of the unit bidiagonal factor L or the conjugated values of the superdiagonal of U from the factorization computed by <code>?pttrf</code> (see $uplo$). The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pttrs` interface are as follows:

d	Holds the vector of length n .
e	Holds the vector of length $(n-1)$.
b	Holds the matrix B of size $(n, nrhs)$.

uplo Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?sytrs

Solves a system of linear equations with a UDU^T - or LDL^T -factored symmetric coefficient matrix.

Syntax

```
call ssytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call sytrs( a, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the system of linear equations $A * X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

if *uplo*='U', $A = U * D * U^T$

if *uplo*='L', $A = L * D * L^T$,

where U and L are upper and lower triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array *ipiv* returned by the factorization routine ?[sytrf](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *a* stores the upper triangular factor U of the factorization $A = U * D * U^T$.
If *uplo* = 'L', the array *a* stores the lower triangular factor L of the factorization $A = L * D * L^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ipiv INTEGER. Array, size at least $\max(1, n)$. The *ipiv* array, as returned by ?[sytrf](#).

a, b REAL for ssytrs

DOUBLE PRECISION for `dsytrs`

COMPLEX for `csytrs`

DOUBLE COMPLEX for `zsytrs`.

Arrays: $a(lda, *)$, $b(ldb, *)$.

The array a contains the factor U or L (see *uplo*). The second dimension of a must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of b must be at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

info INTEGER. If $info=0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrs` interface are as follows:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size $(n, nrhs)$.

ipiv Holds the vector of length n .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T \text{ or } |E| \leq c(n) \varepsilon P |L| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \frac{\| |A^{-1}| |A| |x| \|_{\infty} / \|x\|_{\infty} \leq \|A^{-1}\|_{\infty} \|A\|_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?sycon`.

To refine the solution and estimate the error, call `?sytrfs`.

See Also

Matrix Storage Schemes

`?sytrs_rook`

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
call ssytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call sytrs_rook( a, b, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves a system of linear equations $A * X = B$ with a symmetric matrix A , using the factorization $A = U * D * U^T$ or $A = L * D * L^T$ computed by `?sytrf_rook`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the factorization is of the form $A = U * D * U^T$. If <code>uplo = 'L'</code> , the factorization is of the form $A = L * D * L^T$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<code>ipiv</code>	INTEGER. Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by <code>?sytrf_rook</code> .
<code>a, b</code>	REAL for <code>ssytrs_rook</code> DOUBLE PRECISION for <code>dsytrs_rook</code> COMPLEX for <code>csytrs_rook</code> DOUBLE COMPLEX for <code>zsytrs_rook</code> .

Arrays: $a(lda, *)$, $b(ldb, *)$.

The array a contains the block diagonal matrix D and the multipliers used to obtain U or L as computed by `?sytrf_rook` (see `uplo`).

The array b contains the matrix B whose columns are the right-hand sides for the system of equations.

The second dimension of a must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.

`lda` INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

`ldb` INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

`b` Overwritten by the solution matrix X .

`info` INTEGER. If `info=0`, the execution is successful.
If `info = -i`, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrs_rook` interface are as follows:

`a` Holds the matrix A of size (n, n) .

`b` Holds the matrix B of size $(n, nrhs)$.

`ipiv` Holds the vector of length n .

`uplo` Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hetrs

Solves a system of linear equations with a UDU^T - or LDL^T -factored Hermitian coefficient matrix.

Syntax

```
call chetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

```
call zhetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

```
call hetrs( a, b, ipiv [, uplo] [, info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine solves for X the system of linear equations $A^*X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

$$\text{if } uplo='U', \quad A = U^*D^*U^H$$

$$\text{if } uplo='L', \quad A = L^*D^*L^H,$$

where U and L are upper and lower triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine [?hetrf](#).

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array a stores the upper triangular factor U of the factorization $A = U^*D^*U^H$. If <code>uplo = 'L'</code> , the array a stores the lower triangular factor L of the factorization $A = L^*D^*L^H$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<code>ipiv</code>	INTEGER. Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by ?hetrf .
<code>a, b</code>	COMPLEX for <code>chetrs</code> DOUBLE COMPLEX for <code>zhetrs</code> . Arrays: $a(lda, *)$, $b(l db, *)$. The array a contains the factor U or L (see <code>uplo</code>). The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.
<code>lda</code>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

`b` Overwritten by the solution matrix X .

info INTEGER. If *info*=0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrs` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).
b Holds the matrix *B* of size (*n*, *nrhs*).
ipiv Holds the vector of length *n*.
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x_0\|_\infty} \leq \frac{\|A^{-1}\|_\infty \|E\|_\infty}{1 - \|A^{-1}\|_\infty \|E\|_\infty} \|x_0\|_\infty$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty}{\|x\|_\infty} \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call [?hecon](#).

To refine the solution and estimate the error, call [?herfs](#).

See Also

[Matrix Storage Schemes](#)

[?hetrs_rook](#)

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
call chetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call hetrs_rook( a, b, ipiv [, uplo] [, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for a system of linear equations $A^*X = B$ with a complex Hermitian matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by [?hetrf_rook](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the factorization is of the form $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the factorization is of the form $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf_rook .
<i>a, b</i>	COMPLEX for <code>chetrs_rook</code> DOUBLE COMPLEX for <code>zhetrs_rook</code> . Arrays: $a(lda, *)$, $b(ldb, *)$. The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf_rook (see <i>uplo</i>). The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrs_rook` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

?sytrs2

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
call ssytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call dsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call csytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call sytrs2( a, b, ipiv[, uplo][, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric matrix A using the factorization of A :

$$\text{if } uplo='U', \quad A = U \cdot D \cdot U^T$$

$$\text{if } uplo='L', \quad A = L \cdot D \cdot L^T$$

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a symmetric block-diagonal matrix.

The factorization is computed by `?sytrf`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the upper triangular factor U of the factorization $A = U \cdot D \cdot U^T$. If <code>uplo = 'L'</code> , the array <code>a</code> stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^T$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.

<i>a, b</i>	<p>REAL for <code>ssytrs2</code></p> <p>DOUBLE PRECISION for <code>dsytrs2</code></p> <p>COMPLEX for <code>csytrs2</code></p> <p>DOUBLE COMPLEX for <code>zsytrs2</code></p> <p>Arrays: $a(lda, *)$, $b ldb, *)$.</p> <p>The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?sytrf</code>.</p> <p>The array b contains the right-hand side matrix B.</p> <p>The second dimension of a must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array of size n . The <i>ipiv</i> array contains details of the interchanges and the block structure of D as determined by <code>?sytrf</code> .
<i>work</i>	<p>REAL for <code>ssytrs2</code></p> <p>DOUBLE PRECISION for <code>dsytrs2</code></p> <p>COMPLEX for <code>csytrs2</code></p> <p>DOUBLE COMPLEX for <code>zsytrs2</code></p> <p>Workspace array, size n.</p>

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrs2` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Indicates how the input matrix A has been factored. Must be 'U' or 'L'.

See Also

[?sytrf](#)

Matrix Storage Schemes

?hetrs2

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
call chetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zhetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call hetrs2( a, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves a system of linear equations $A^*X = B$ with a complex Hermitian matrix A using the factorization of A :

if $uplo='U'$, $A = U^*D^*U^H$

if $uplo='L'$, $A = L^*D^*L^H$

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a Hermitian block-diagonal matrix.

The factorization is computed by ?hetrf.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U^*D^*U^H$.
If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L^*D^*L^H$.

n INTEGER. The order of matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, b COMPLEX for chetrs2
DOUBLE COMPLEX for zhetrs2
Arrays: $a(lda, *)$, $b(ldb, *)$.
The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf.
The array b contains the right-hand side matrix B .
The second dimension of a must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array of size <i>n</i> . The <i>ipiv</i> array contains details of the interchanges and the block structure of <i>D</i> as determined by ?hetrf.
<i>work</i>	COMPLEX for chetrs2 DOUBLE COMPLEX for zhetrs2 Workspace array, size <i>n</i> .

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrs2` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

[?hetrf](#)

[Matrix Storage Schemes](#)

?sptrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix using packed storage.

Syntax

```
call ssptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call sptrs( ap, b, ipiv [, uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the system of linear equations $A \cdot X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

if $uplo='U'$, $A = U \cdot D \cdot U^T$

if $uplo='L'$, $A = L \cdot D \cdot L^T$,

where U and L are upper and lower *packed* triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply the factor U (or L) and the array $ipiv$ returned by the factorization routine [?spturf](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array ap stores the packed factor U of the factorization $A = U \cdot D \cdot U^T$. If $uplo = 'L'$, the array ap stores the packed factor L of the factorization $A = L \cdot D \cdot L^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by ?spturf .
<i>ap</i>	REAL for <code>ssptrs</code> DOUBLE PRECISION for <code>dsptrs</code> COMPLEX for <code>csptrs</code> DOUBLE COMPLEX for <code>zsptrs</code> . The dimension of array ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by $uplo$, in <i>packed storage</i> (see Matrix Storage Schemes).
<i>b</i>	REAL for <code>ssptrs</code> DOUBLE PRECISION for <code>dsptrs</code> COMPLEX for <code>csptrs</code> DOUBLE COMPLEX for <code>zsptrs</code> . The array $b(l_{db}, *)$ contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spttrs` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon P|U||D||U^T|P^T \text{ or } |E| \leq c(n)\epsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of *n*, and ϵ is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\epsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?spcon`.

To refine the solution and estimate the error, call `?sprfs`.

See Also

Matrix Storage Schemes

?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix using packed storage.

Syntax

```
call chptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

```
call zhptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

```
call hptrs( ap, b, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the system of linear equations $A^*X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

if $uplo='U'$, $A = U^*D^*U^H$

if $uplo='L'$, $A = L^*D^*L^H$,

where U and L are upper and lower *packed* triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

You must supply to this routine the arrays ap (containing U or L) and $ipiv$ in the form returned by the factorization routine [?hptrf](#).

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array ap stores the packed factor U of the factorization $A = U^*D^*U^H$. If $uplo = 'L'$, the array ap stores the packed factor L of the factorization $A = L^*D^*L^H$.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
$ipiv$	INTEGER. Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by ?hptrf .
ap	COMPLEX for <code>chptrs</code> DOUBLE COMPLEX for <code>zhptrs</code> . The dimension of array $ap(*)$ must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by $uplo$, in <i>packed storage</i> (see Matrix Storage Schemes).
b	COMPLEX for <code>chptrs</code> DOUBLE COMPLEX for <code>zhptrs</code> . The array $b(ldb, *)$ contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix *X*.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hptrs` interface are as follows:

ap Holds the array *A* of size $(n * (n+1) / 2)$.
b Holds the matrix *B* of size $(n, nrhs)$.
ipiv Holds the vector of length *n*.
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^H| P^T \text{ or } |E| \leq c(n) \varepsilon P |L| |D| |L^H| P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty}{\|x\|_\infty} \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?hpcon](#).

To refine the solution and estimate the error, call [?hprfs](#).

See Also

[Matrix Storage Schemes](#)

[?trtrs](#)

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call strtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call dtrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ctrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ztrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call trtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the following systems of linear equations with a triangular matrix A , with multiple right-hand sides stored in B :

$A^*X = B$ if $trans = 'N'$,
 $A^T * X = B$ if $trans = 'T'$,
 $A^H * X = B$ if $trans = 'C'$ (for complex matrices only).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Indicates whether A is upper or lower triangular:
 If $uplo = 'U'$, then A is upper triangular.
 If $uplo = 'L'$, then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
 If $trans = 'N'$, then $A^*X = B$ is solved for X .
 If $trans = 'T'$, then $A^T * X = B$ is solved for X .
 If $trans = 'C'$, then $A^H * X = B$ is solved for X .

diag CHARACTER*1. Must be 'N' or 'U'.
 If $diag = 'N'$, then A is not a unit triangular matrix.
 If $diag = 'U'$, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a .

n INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a REAL for strtrs
 DOUBLE PRECISION for dtrtrs
 COMPLEX for ctrtrs
 DOUBLE COMPLEX for ztrtrs.
 The array $a(lda,*)$ contains the matrix A .

The second dimension of a must be at least $\max(1, n)$.

b REAL for `strtrs`
 DOUBLE PRECISION for `dtrtrs`
 COMPLEX for `ctrtrs`
 DOUBLE COMPLEX for `ztrtrs`.

The array $b(ldb, *)$ contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

$info$ INTEGER. If $info=0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trtrs` interface are as follows:

a Stands for argument ap in FORTRAN 77 interface. Holds the matrix A of size $(n*(n+1)/2)$.

b Holds the matrix B of size $(n, nrhs)$.

$uplo$ Must be 'U' or 'L'. The default value is 'U'.

$trans$ Must be 'N', 'C', or 'T'. The default value is 'N'.

$diag$ Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon \text{ provided } c(n) \operatorname{cond}(A, x) \varepsilon < 1$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?trcon`.

To estimate the error in the solution, call `?trrfs`.

See Also

Matrix Storage Schemes

`?tptrs`

Solves a system of linear equations with a packed triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call stptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call dtptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ctptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call ztptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call tptrs( ap, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the following systems of linear equations with a packed triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Input Parameters

`uplo` CHARACTER*1. Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If `uplo = 'U'`, then A is upper triangular.
If `uplo = 'L'`, then A is lower triangular.

<i>trans</i>	<p>CHARACTER*1. Must be 'N' or 'T' or 'C'.</p> <p>If <i>trans</i> = 'N', then $A * X = B$ is solved for X.</p> <p>If <i>trans</i> = 'T', then $A^T * X = B$ is solved for X.</p> <p>If <i>trans</i> = 'C', then $A^H * X = B$ is solved for X.</p>
<i>diag</i>	<p>CHARACTER*1. Must be 'N' or 'U'.</p> <p>If <i>diag</i> = 'N', then A is not a unit triangular matrix.</p> <p>If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i>.</p>
<i>n</i>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap</i>	<p>REAL for <i>stptrs</i></p> <p>DOUBLE PRECISION for <i>dtptrs</i></p> <p>COMPLEX for <i>ctptrs</i></p> <p>DOUBLE COMPLEX for <i>ztptrs</i>.</p> <p>The dimension of array <i>ap</i>(*) must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the matrix A in <i>packed storage</i> (see Matrix Storage Schemes).</p>
<i>b</i>	<p>REAL for <i>stptrs</i></p> <p>DOUBLE PRECISION for <i>dtptrs</i></p> <p>COMPLEX for <i>ctptrs</i></p> <p>DOUBLE COMPLEX for <i>ztptrs</i>.</p> <p>The array <i>b</i>(<i>ldb</i>,*) contains the matrix B whose columns are the right-hand sides for the system of equations.</p> <p>The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p>
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	<p>INTEGER. If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *tpttrs* interface are as follows:

<i>ap</i>	Holds the array A of size $(n * (n+1) / 2)$.
-----------	---

<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x_0\|_\infty} \leq \text{cond}(A, x) \frac{\|E\|_\infty}{\|A\|_\infty}$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty}{\|x\|_\infty} \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tpcon](#).

To estimate the error in the solution, call [?tprfs](#).

See Also

Matrix Storage Schemes

[?tbtrs](#)

Solves a system of linear equations with a band triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
call stbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call dtbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ctbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ztbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call tbtrs( ab, b [,uplo] [, trans] [,diag] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the following systems of linear equations with a band triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$ if $trans = 'N'$,
 $A^T * X = B$ if $trans = 'T'$,
 $A^H * X = B$ if $trans = 'C'$ (for complex matrices only).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Indicates whether A is upper or lower triangular:
 If $uplo = 'U'$, then A is upper triangular.
 If $uplo = 'L'$, then A is lower triangular.

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
 If $trans = 'N'$, then $A * X = B$ is solved for X .
 If $trans = 'T'$, then $A^T * X = B$ is solved for X .
 If $trans = 'C'$, then $A^H * X = B$ is solved for X .

diag CHARACTER*1. Must be 'N' or 'U'.
 If $diag = 'N'$, then A is not a unit triangular matrix.
 If $diag = 'U'$, then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array ab .

n INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.

kd INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ab REAL for stbtrs
 DOUBLE PRECISION for dtbtrs
 COMPLEX for ctbtrs
 DOUBLE COMPLEX for ztbtrs.
 The array $ab(ldab, *)$ contains the matrix A in *band storage* form.
 The second dimension of ab must be at least $\max(1, n)$.

b REAL for stbtrs
 DOUBLE PRECISION for dtbtrs
 COMPLEX for ctbtrs
 DOUBLE COMPLEX for ztbtrs.

The array $b(l_{db}, *)$ contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of b at least $\max(1, nrhs)$.

ldab INTEGER. The leading dimension of ab ; $ldab \geq kd + 1$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

info INTEGER. If $info=0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tbtrs` interface are as follows:

- ab* Holds the array A of size $(kd+1, n)$
- b* Holds the matrix B of size $(n, nrhs)$.
- uplo* Must be 'U' or 'L'. The default value is 'U'.
- trans* Must be 'N', 'C', or 'T'. The default value is 'N'.
- diag* Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon|A|$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\|x - x_0\|_{\infty} \leq \frac{\|A^{-1}\|_{\infty} \|A\|_{\infty} \|x\|_{\infty}}{\text{cond}(A, x)} \leq \|A^{-1}\|_{\infty} \|A\|_{\infty} = \kappa_{\infty}(A)$$

where $\text{cond}(A, x) = \frac{\|A^{-1}\|_{\infty} \|A\|_{\infty} \|x\|_{\infty}}{\|x - x_0\|_{\infty}} \leq \|A^{-1}\|_{\infty} \|A\|_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n \cdot kd$ for real flavors and $8n \cdot kd$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?tbcon`.

To estimate the error in the solution, call `?tbrfs`.

See Also

[Matrix Storage Schemes](#)

Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see [Error Analysis](#)). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

?gecon

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

```
call sgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call gecon( a, anorm, rcond [,norm] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a general matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_{\infty}(A^T) = \kappa_{\infty}(A^H)$$

$$\kappa_{\infty}(A) = \|A\|_{\infty} \|A^{-1}\|_{\infty} = \kappa_1(A^T) = \kappa_1(A^H).$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_{\infty} = \max_i \sum_j |a_{ij}|$)
- call `?getrf` to compute the *LU* factorization of A .

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.

<i>a, work</i>	<p>REAL for <code>sgecon</code></p> <p>DOUBLE PRECISION for <code>dgecon</code></p> <p>COMPLEX for <code>cgecon</code></p> <p>DOUBLE COMPLEX for <code>zgecon</code>. Arrays: $a(lda, *)$, $work(*)$.</p> <p>The array a contains the LU-factored matrix A, as returned by ?getrf. The second dimension of a must be at least $\max(1, n)$. The array $work$ is a workspace for the routine.</p> <p>The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.</p>
<i>anorm</i>	<p>REAL for single precision flavors.</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>The norm of the <i>original</i> matrix A (see Description).</p>
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	<p>REAL for <code>cgecon</code></p> <p>DOUBLE PRECISION for <code>zgecon</code>.</p> <p>Workspace array, size at least $\max(1, 2*n)$.</p>

Output Parameters

<i>rcond</i>	<p>REAL for single precision flavors.</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.</p>
<i>info</i>	<p>INTEGER. If $info=0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gecon` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n^2$ floating-point operations for real flavors and $8*n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?gbcon

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

```
call sgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call dgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call cgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
call zgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
call gbcon( ab, ipiv, anorm, rcond [,kl] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a general band matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?gbtrf](#) to compute the LU factorization of A .

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.

<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . ($ldab \geq 2*kl + ku + 1$).
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gbtrf .
<i>ab, work</i>	REAL for sgbcon DOUBLE PRECISION for dgbcon COMPLEX for cgbcon DOUBLE COMPLEX for zgbcon . Arrays: <i>ab</i> (<i>ldab</i> ,*), <i>work</i> (*). The array <i>ab</i> contains the factored band matrix <i>A</i> , as returned by ?gbtrf . The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see Description).
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for cgbcon DOUBLE PRECISION for zgbcon . Workspace array, size at least $\max(1, 2*n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbccon` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(2*kl+ku+1, n)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>norm</i>	Must be '1', '0', or 'I'. The default value is '1'.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda-2*kl-1$.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(ku + 2kl)$ floating-point operations for real flavors and $8n(ku + 2kl)$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?gtcon

Estimates the reciprocal of the condition number of a tridiagonal matrix.

Syntax

```
call sgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call dgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call cgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call gtcon( dl, d, du, du2, ipiv, anorm, rcond [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix *A* in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?gttrf](#) to compute the *LU* factorization of *A*.

Input Parameters

norm CHARACTER*1. Must be '1' or '0' or 'I'.

If $norm = '1'$ or $'0'$, then the routine estimates the condition number of matrix A in 1-norm.

If $norm = 'I'$, then the routine estimates the condition number of matrix A in infinity-norm.

n

INTEGER. The order of the matrix A ; $n \geq 0$.

$dl, d, du, du2$

REAL for sgtcon

DOUBLE PRECISION for dgtcon

COMPLEX for cgtcon

DOUBLE COMPLEX for zgtcon.

Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$, $du2(n-2)$.

The array dl contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A as computed by [?gttrf](#).

The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

The array du contains the $(n-1)$ elements of the first superdiagonal of U .

The array $du2$ contains the $(n-2)$ elements of the second superdiagonal of U .

$ipiv$

INTEGER.

Array, size (n) . The array of pivot indices, as returned by [?gttrf](#).

$anorm$

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

$work$

REAL for sgtcon

DOUBLE PRECISION for dgtcon

COMPLEX for cgtcon

DOUBLE COMPLEX for zgtcon.

Workspace array, size $(2*n)$.

$iwork$

INTEGER. Workspace array, size (n) . Used for real flavors only.

Output Parameters

$rcond$

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond=0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

<i>dl</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
call spocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call pocon( a, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?potrf](#) to compute the Cholesky factorization of *A*.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a, work</i>	REAL for spocon DOUBLE PRECISION for dpocon COMPLEX for cpocon DOUBLE COMPLEX for zpocon. Arrays: <i>a</i> (<i>lda</i> , *), <i>work</i> (*). The array <i>a</i> contains the factored matrix <i>A</i> , as returned by ?potrf . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>anorm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for cpocon DOUBLE PRECISION for zpocon. Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pocon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?ppcon

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

```
call sppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call ppcon( ap, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?pptrf](#) to compute the Cholesky factorization of A .

Input Parameters

<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.
<code>ap, work</code>	REAL for <code>sppcon</code> DOUBLE PRECISION for <code>dppcon</code> COMPLEX for <code>cppcon</code> DOUBLE COMPLEX for <code>zppcon</code> .

Arrays: $ap(*)$, $work(*)$.

The array ap contains the packed factored matrix A , as returned by [?pptrf](#). The dimension of ap must be at least $\max(1, n(n+1)/2)$.

The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

$anorm$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

$iwork$

INTEGER. Workspace array, size at least $\max(1, n)$.

$rwork$

REAL for `cppcon`

DOUBLE PRECISION for `zppcon`.

Workspace array, size at least $\max(1, n)$.

Output Parameters

$rcond$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ppcon` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

$uplo$

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
call spbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call dpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call cpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call zpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call pbcon( ab, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?pbtrf](#) to compute the Cholesky factorization of A .

Input Parameters

n	INTEGER. The order of the matrix A ; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
$ldab$	INTEGER. The leading dimension of the array ab . ($ldab \geq kd + 1$).
$ab, work$	REAL for spbcon DOUBLE PRECISION for dpbcon COMPLEX for cpbcon DOUBLE COMPLEX for zpbcon. Arrays: $ab(ldab, *)$, $work(*)$. The array ab contains the factored matrix A in band form, as returned by ?pbtrf . The second dimension of ab must be at least $\max(1, n)$. The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
$anorm$	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

iwork INTEGER. Workspace array, size at least $\max(1, n)$.

rwork REAL for `cpbcon`
DOUBLE PRECISION for `zpbcon`.
Workspace array, size at least $\max(1, n)$.

Output Parameters

rcond REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbcon` interface are as follows:

ab Holds the array A of size $(kd+1, n)$.
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n(kd + 1)$ floating-point operations for real flavors and $16*n(kd + 1)$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
call sptcon( n, d, e, anorm, rcond, work, info )
call dptcon( n, d, e, anorm, rcond, work, info )
call cptcon( n, d, e, anorm, rcond, work, info )
call zptcon( n, d, e, anorm, rcond, work, info )
```

call `ptcon(d, e, anorm, rcond [,info])`

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L^*D^*L^T$ for real flavors and $A = L^*D^*L^H$ for complex flavors or $A = U^T*D^*U$ for real flavors and $A = U^H*D^*U$ for complex flavors computed by `?pttrf` :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

The norm $\|A^{-1}\|_1$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* as $\|A\|_1 = \max_j \sum_i |a_{ij}|$
- call `?pttrf` to compute the factorization of *A*.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>d, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, dimension (<i>n</i>). The array <i>d</i> contains the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization of <i>A</i> , as computed by <code>?pttrf</code> ; <i>work</i> is a workspace array.
<i>e</i>	REAL for <code>sptcon</code> DOUBLE PRECISION for <code>dptcon</code> COMPLEX for <code>cptcon</code> DOUBLE COMPLEX for <code>zptcon</code> . Array, size (<i>n</i> -1). Contains off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by <code>?pttrf</code> .
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The 1- norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
--------------	---

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

d Holds the vector of length n .

e Holds the vector of length $(n-1)$.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n*(kd + 1)$ floating-point operations for real flavors and $16*n*(kd + 1)$ for complex flavors.

?sycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
call ssycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call sycon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mk1.fi, lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?sytrf` to compute the factorization of *A*.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates how the input matrix <i>A</i> has been factored:</p> <p>If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U^*D^*U^T$.</p> <p>If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L^*D^*L^T$.</p>
<i>n</i>	<p>INTEGER. The order of matrix <i>A</i>; $n \geq 0$.</p>
<i>a, work</i>	<p>REAL for <code>ssycon</code></p> <p>DOUBLE PRECISION for <code>dsycon</code></p> <p>COMPLEX for <code>csycon</code></p> <p>DOUBLE COMPLEX for <code>zsycon</code>.</p> <p>Arrays: <i>a</i>(<i>lda</i>,*), <i>work</i>(*).</p> <p>The array <i>a</i> contains the factored matrix <i>A</i>, as returned by <code>?sytrf</code>. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p>The array <i>work</i> is a workspace for the routine.</p> <p>The dimension of <i>work</i> must be at least $\max(1, 2*n)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of <i>a</i>; $lda \geq \max(1, n)$.</p>
<i>ipiv</i>	<p>INTEGER. Array, size at least $\max(1, n)$.</p> <p>The array <i>ipiv</i>, as returned by <code>?sytrf</code>.</p>
<i>anorm</i>	<p>REAL for single precision flavors.</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).</p>
<i>iwork</i>	<p>INTEGER. Workspace array, size at least $\max(1, n)$.</p>

Output Parameters

<i>rcond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.</p>
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sycon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?sycon_rook`

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
call ssycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call sycon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?sytrf_rook` to compute the factorization of A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <code>uplo = 'U'</code> , the array <code>a</code> stores the upper triangular factor U of the factorization $A = U*D*U^T$.

If `uplo = 'L'`, the array `a` stores the lower triangular factor L of the factorization $A = L * D * L^T$.

`n` INTEGER. The order of matrix A ; $n \geq 0$.

`a, work` REAL for `ssycon_rook`
 DOUBLE PRECISION for `dsycon_rook`
 COMPLEX for `csycon_rook`
 DOUBLE COMPLEX for `zsycon_rook`.

Arrays: `a(lda,*)`, `work(*)`.

The array `a` contains the factored matrix A , as returned by `?sytrf_rook`. The second dimension of `a` must be at least $\max(1, n)$.

The array `work` is a workspace for the routine.

The dimension of `work` must be at least $\max(1, 2 * n)$.

`lda` INTEGER. The leading dimension of `a`; $lda \geq \max(1, n)$.

`ipiv` INTEGER. Array, size at least $\max(1, n)$.

The array `ipiv`, as returned by `?sytrf_rook`.

`anorm` REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

`iwork` INTEGER. Workspace array, size at least $\max(1, n)$.

Output Parameters

`rcond` REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets `rcond = 0` if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info` INTEGER. If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sycon_rook` interface are as follows:

`a` Holds the matrix A of size (n, n) .

<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

```
call checon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call hecon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?hetrf](#) to compute the factorization of A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U*D*U^H$. If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L*D*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for checon DOUBLE COMPLEX for zhecon. Arrays: $a(lda, *)$, $work(*)$.

The array *a* contains the factored matrix *A*, as returned by [?hetrf](#). The second dimension of *a* must be at least $\max(1, n)$.

The array *work* is a workspace for the routine. The dimension of *work* must be at least $\max(1, 2*n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv INTEGER. Array, size at least $\max(1, n)$.

The array *ipiv*, as returned by [?hetrf](#).

anorm REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
The norm of the *original* matrix *A* (see *Description*).

Output Parameters

rcond REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hecon` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

ipiv Holds the vector of length *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also
[Matrix Storage Schemes](#)

?hecon_rook

Estimates the reciprocal of the condition number of a Hermitian matrix using factorization obtained with one of the bounded diagonal pivoting methods (max 2 interchanges).

Syntax

```
call checon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call hecon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix A using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by [hetrf_rook](#).

An estimate is obtained for $\text{norm}(A^{-1})$, and the reciprocal of the condition number is computed as $rcond = 1/(\text{anorm}*\text{norm}(A^{-1}))$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor U of the factorization $A = U*D*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor L of the factorization $A = L*D*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for checon_rook COMPLEX*16 for zhecon_rook . Arrays: <i>a</i> (<i>lda</i> , <i>n</i>), <i>work</i> (*). The array <i>a</i> contains the factored matrix A , as returned by ? hetrf_rook . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The array <i>ipiv</i> , as returned by hetrf_rook .
<i>anorm</i>	REAL for checon_rook DOUBLE PRECISION for zhecon_rook . The 1-norm of the original matrix A (see Description).

Output Parameters

<i>rcond</i>	REAL for checon_rook DOUBLE PRECISION for zhecon_rook. The reciprocal of the condition number of the matrix <i>A</i> , computed as $rcond = 1/(anorm * ainvnm)$, where <i>ainvnm</i> is an estimate of the 1-norm of A^{-1} computed in this routine.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hecon_rook` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax

```
call sspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call spcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a packed symmetric matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)

- call `?sptf` to compute the factorization of A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array <code>ap</code> stores the packed upper triangular factor U of the factorization $A = U^*D^*U^T$. If <code>uplo</code> = 'L', the array <code>ap</code> stores the packed lower triangular factor L of the factorization $A = L^*D^*L^T$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>ap, work</code>	REAL for <code>sspcon</code> DOUBLE PRECISION for <code>dspcon</code> COMPLEX for <code>cspcon</code> DOUBLE COMPLEX for <code>zspcon</code> . Arrays: <code>ap(*)</code> , <code>work(*)</code> . The array <code>ap</code> contains the packed factored matrix A , as returned by <code>?sptf</code> . The dimension of <code>ap</code> must be at least $\max(1, n(n+1)/2)$. The array <code>work</code> is a workspace for the routine. The dimension of <code>work</code> must be at least $\max(1, 2*n)$.
<code>ipiv</code>	INTEGER. Array, size at least $\max(1, n)$. The array <code>ipiv</code> , as returned by <code>?sptf</code> .
<code>anorm</code>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).
<code>iwork</code>	INTEGER. Workspace array, size at least $\max(1, n)$.

Output Parameters

<code>rcond</code>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <code>rcond</code> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <code>rcond</code> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spcon` interface are as follows:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

```
call chpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zhpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call hpcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A)\text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute `anorm` (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?hptrf` to compute the factorization of A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <code>uplo = 'U'</code> , the array <code>ap</code> stores the packed upper triangular factor U of the factorization $A = U^*D^*U^T$.

If `uplo = 'L'`, the array `ap` stores the packed lower triangular factor L of the factorization $A = L*D*L^T$.

`n` INTEGER. The order of matrix A ; $n \geq 0$.

`ap, work` COMPLEX for `chpcon`
DOUBLE COMPLEX for `zhpcon`.

The array `ap(*)` contains the packed factored matrix A , as returned by `?hptraf`. The dimension of `ap` must be at least $\max(1, n(n+1)/2)$.

The array `work(*)` is a workspace for the routine. The dimension of `work` must be at least $\max(1, 2*n)$.

`ipiv` INTEGER.

Array, size at least $\max(1, n)$. The array `ipiv`, as returned by `?hptraf`.

`anorm` REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

Output Parameters

`rcond` REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets `rcond = 0` if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info` INTEGER. If `info = 0`, the execution is successful.
If `info = -i`, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbcon` interface are as follows:

`ap` Holds the array A of size $(n*(n+1)/2)$.

`ipiv` Holds the vector of length n .

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

Matrix Storage Schemes

strcon

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

```
call strcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call dtrcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call ctrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call ztrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call trcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangle of A , other array elements are not referenced. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangle of A , other array elements are not referenced.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for strcon DOUBLE PRECISION for dtrcon

COMPLEX for `ctrcon`

DOUBLE COMPLEX for `ztrcon`.

The array $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

The array $work(*)$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

iwork INTEGER. Workspace array, size at least $\max(1, n)$.

rwork REAL for `ctrcon`

DOUBLE PRECISION for `ztrcon`.

Workspace array, size at least $\max(1, n)$.

Output Parameters

rcond REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trcon` interface are as follows:

a Holds the matrix A of size (n, n) .

norm Must be '1', 'O', or 'I'. The default value is '1'.

uplo Must be 'U' or 'L'. The default value is 'U'.

diag Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

Matrix Storage Schemes

stpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

```
call stpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call dtpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call ctpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call ztpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call tpcon( ap, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a packed triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap, work</i>	REAL for stpcon DOUBLE PRECISION for dtpcon

COMPLEX for `ctpcon`

DOUBLE COMPLEX for `ztpcon`.

The array `ap(*)` contains the packed matrix A . The dimension of `ap` must be at least $\max(1, n(n+1)/2)$.

The array `work(*)` is a workspace for the routine. The dimension of `work` must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

`iwork` INTEGER. Workspace array, size at least $\max(1, n)$.

`rwork` REAL for `ctpcon`
DOUBLE PRECISION for `ztpcon`.
Workspace array, size at least $\max(1, n)$.

Output Parameters

`rcond` REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal of the condition number. The routine sets `rcond` = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info` INTEGER. If `info` = 0, the execution is successful.
If `info` = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tpcon` interface are as follows:

`ap` Holds the array A of size $(n*(n+1)/2)$.
`norm` Must be '1', 'O', or 'I'. The default value is '1'.
`uplo` Must be 'U' or 'L'. The default value is 'U'.
`diag` Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

Matrix Storage Schemes

stbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

```
call stbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call tbcon( ab, rcond [,uplo] [,diag] [,norm] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>ab, work</i>	REAL for stbcon DOUBLE PRECISION for dtbcon

COMPLEX for `ctbcon`

DOUBLE COMPLEX for `ztbcon`.

The array `ab(ldab,*)` contains the band matrix A . The second dimension of `ab` must be at least $\max(1, n)$.

The array `work` is a workspace for the routine. The dimension of `work(*)` must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

`ldab` INTEGER. The leading dimension of the array `ab`. ($ldab \geq kd + 1$).

`iwork` INTEGER. Workspace array, size at least $\max(1, n)$.

`rwork` REAL for `ctbcon`

DOUBLE PRECISION for `ztbcon`.

Workspace array, size at least $\max(1, n)$.

Output Parameters

`rcond` REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets `rcond` = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

`info` INTEGER. If `info` = 0, the execution is successful.

If `info` = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tbcon` interface are as follows:

`ab` Holds the array A of size $(kd+1, n)$.

`norm` Must be '1', 'O', or 'I'. The default value is '1'.

`uplo` Must be 'U' or 'L'. The default value is 'U'.

`diag` Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n(kd + 1)$ floating-point operations for real flavors and $8*n(kd + 1)$ operations for complex flavors.

See Also

Matrix Storage Schemes

Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Routines for Solving Systems of Linear Equations](#)).

?gerfs

Refines the solution of a system of linear equations with a general coefficient matrix and estimates its error.

Syntax

```
call sgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
```

```
call dgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
```

```
call cgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call zgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call gerfs( a, af, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a general matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?getrf](#)
- call the solver routine [?getrs](#).

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', the system has the form $A*X = B$.

If *trans* = 'T', the system has the form $A^T*X = B$.

If $trans = 'C'$, the system has the form $A^H X = B$.

n INTEGER. The order of the matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, af, b, x, work

REAL for sgerfs
 DOUBLE PRECISION for dgerfs
 COMPLEX for cgerfs
 DOUBLE COMPLEX for zgerfs.

Arrays:

a(size *lda* by *) contains the original matrix A , as supplied to [?getrf](#).

af(size *ldaf* by *) contains the factored matrix A , as returned by [?getrf](#).

b(size *ldb* by *) contains the right-hand side matrix B .

x(size *ldx* by *) contains the solution matrix X .

work(size *) is a workspace array.

The second dimension of *a* and *af* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$.

The *ipiv* array, as returned by [?getrf](#).

iwork INTEGER.

Workspace array, size at least $\max(1, n)$.

rwork REAL for cgerfs

DOUBLE PRECISION for zgerfs.

Workspace array, size at least $\max(1, n)$.

Output Parameters

x The refined solution matrix X .

ferr, berr REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gerfs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>af</i>	Holds the matrix <i>AF</i> of size (<i>n</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$ with the same coefficient matrix *A* and different right hand sides *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?gerfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a general coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call sgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
```

```
call dgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
```

```
call cgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

```
call zgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, ldx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed`, `r`, and `c` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<code>trans</code>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <code>trans</code> = 'N', the system has the form $A * X = B$ (No transpose).</p> <p>If <code>trans</code> = 'T', the system has the form $A^T * X = B$ (Transpose).</p> <p>If <code>trans</code> = 'C', the system has the form $A^H * X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).</p>
<code>equed</code>	<p>CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.</p> <p>Specifies the form of equilibration that was done to A before calling this routine.</p> <p>If <code>equed</code> = 'N', no equilibration was done.</p> <p>If <code>equed</code> = 'R', row equilibration was done, that is, A has been premultiplied by $diag(r)$.</p> <p>If <code>equed</code> = 'C', column equilibration was done, that is, A has been postmultiplied by $diag(c)$.</p> <p>If <code>equed</code> = 'B', both row and column equilibration was done, that is, A has been replaced by $diag(r) * A * diag(c)$. The right-hand side B has been changed accordingly.</p>
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

a, af, b, work REAL for sgerfsx
 DOUBLE PRECISION for dgerfsx
 COMPLEX for cgerfsx
 DOUBLE COMPLEX for zgerfsx.
 Arrays: *a* (size *lda* by *), *af* (size *ldaf* by *), *b*(size *ldb* by *), *work*(*).
 The array *a* contains the original *n*-by-*n* matrix *A*.
 The array *af* contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization $A = P*U*L$ as computed by ?getrf.
 The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.
work (size *) is a workspace array. The dimension of *work* must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ipiv INTEGER.
 Array, size at least $\max(1, n)$. Contains the pivot indices as computed by ?getrf; for row $1 \leq i \leq n$, row *i* of the matrix was interchanged with row *ipiv*(*i*).

r, c REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*.
equed = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.
 If *equed* = 'R' or 'B', each element of *r* must be positive.
 If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.
 If *equed* = 'C' or 'B', each element of *c* must be positive.
 Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; $ldb \geq \max(1, n)$.

x REAL for sgerfsx
 DOUBLE PRECISION for dgerfsx
 COMPLEX for cgerfsx
 DOUBLE COMPLEX for zgerfsx.

Array, size ldx by $*$.

The solution matrix X as computed by `?getrs`

<code>ldx</code>	INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.
<code>n_err_bnds</code>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below.
<code>nparams</code>	INTEGER. Specifies the number of parameters set in <code>params</code> . If ≤ 0 , the <code>params</code> array is never referenced and default values are used.
<code>params</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <code>nparams</code> . Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <code>nparams</code> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <code>nparams = 0</code> , which prevents the source code from accessing the <code>params</code> argument. <code>params(1)</code> : Whether to perform iterative refinement or not. Default: 1.0 =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision. (Other values are reserved for future use.) <code>params(2)</code> : Maximum number of residual computations allowed for refinement. Default 10.0 Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy. <code>params(3)</code> : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).
<code>iwork</code>	INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.
<code>rwork</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

<i>x</i>	<p>REAL for <code>sgerfsx</code></p> <p>DOUBLE PRECISION for <code>dgerfsx</code></p> <p>COMPLEX for <code>cgerfsx</code></p> <p>DOUBLE COMPLEX for <code>zgerfsx</code>.</p> <p>The improved solution matrix <i>X</i>.</p>		
<i>rcond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.</p>		
<i>berr</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>		
<i>err_bnds_norm</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array of size <i>nrhs</i> by <i>n_err_bnds</i>. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:</p> <p>Normwise relative error in the <i>i</i>-th solution vector</p> $\frac{\max_j X_{true_{ji}} - X_{ji} }{\max_j X_{ji} }$ <p>The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.</p> <p>The first index in <code>err_bnds_norm(i, :)</code> corresponds to the <i>i</i>-th right-hand side.</p> <p>The second index in <code>err_bnds_norm(:, err)</code> contains the following three fields:</p> <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 20px;"><i>err</i>=1</td> <td>"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.</td> </tr> </table>	<i>err</i> =1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<i>err</i> =1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.		

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params*(3) = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first (*:,n_err_bnds*) entries are returned.

The first index in *err_bnds_comp*(*i*,:) corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp*(:,*err*) contains the following three fields:

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

params REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Output parameter only if the input contains erroneous values, namely, in *params(1)*, *params(2)*, *params(3)*. In such a case, the corresponding elements of *params* are filled with default values on output.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.
If *info* = $n + j$: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that $\text{err_bnds_norm}(j, 1) =$

0.0 or `err_bnds_comp(j,1) = 0.0`). See the definition of `err_bnds_norm` and `err_bnds_comp` for `err = 1`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

?gbrfs

Refines the solution of a system of linear equations with a general band coefficient matrix and estimates its error.

Syntax

```
call sgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
```

```
call dgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
```

```
call cgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

```
call zgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

```
call gbrfs( ab, afb, ipiv, b, x [,kl] [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$ with a band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gbtrf](#)
- call the solver routine [?gbtrs](#).

Input Parameters

`trans` CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If `trans = 'N'`, the system has the form $A^*X = B$.

If `trans = 'T'`, the system has the form $A^T * X = B$.

If `trans = 'C'`, the system has the form $A^H * X = B$.

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for <code>sgbrfs</code> DOUBLE PRECISION for <code>dgbrfs</code> COMPLEX for <code>cgbrfs</code> DOUBLE COMPLEX for <code>zgbrfs</code> . Arrays: <i>ab</i> (size <i>ldab</i> by *) contains the original band matrix <i>A</i> , as supplied to <code>?gbtrf</code> , but stored in rows from 1 to $kl + ku + 1$. <i>afb</i> (size <i>ldafb</i> by *) contains the factored band matrix <i>A</i> , as returned by <code>?gbtrf</code> . <i>b</i> (size <i>ldb</i> by *) contains the right-hand side matrix <i>B</i> . <i>x</i> (size <i>ldx</i> by *) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The second dimension of <i>ab</i> and <i>afb</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> .
<i>ldafb</i>	INTEGER. The leading dimension of <i>afb</i> .
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by <code>?gbtrf</code> .
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>cgbrfs</code> DOUBLE PRECISION for <code>zgbrfs</code> . Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbrfs` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(kl+ku+1, n)$.
<i>afb</i>	Holds the array <i>AF</i> of size $(2*kl*ku+1, n)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda-kl-1$.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?gbrfsx

*Uses extra precise iterative refinement to improve the solution to the system of linear equations with a banded coefficient matrix *A* and provides error bounds and backward error estimates.*

Syntax

```
call sgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call dgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call cgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A^*X = B$ (No transpose).</p> <p>If <i>trans</i> = 'T', the system has the form $A^T * X = B$ (Transpose).</p> <p>If <i>trans</i> = 'C', the system has the form $A^H * X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).</p>
<i>equed</i>	<p>CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.</p> <p>Specifies the form of equilibration that was done to <i>A</i> before calling this routine.</p> <p>If <i>equed</i> = 'N', no equilibration was done.</p> <p>If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag(r)</i>.</p> <p>If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i>.</p> <p>If <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by $diag(r) * A * diag(c)$. The right-hand side <i>B</i> has been changed accordingly.</p>

<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>ab,afb,b,work</i>	<p>REAL for <code>sgbrfsx</code> DOUBLE PRECISION for <code>dgbrfsx</code> COMPLEX for <code>cgbrfsx</code> DOUBLE COMPLEX for <code>zgbrfsx</code>.</p> <p>Arrays: <i>ab</i>(<i>ldab</i>,*), <i>afb</i>(<i>ldafb</i>,*), <i>b</i>(<i>ldb</i>,*), <i>work</i>(*).</p> <p>The array <i>ab</i> contains the original matrix <i>A</i> in band storage, in rows 1 to <i>kl</i> + <i>ku</i> + 1. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows:</p> $ab(ku+1+i-j, j) = A(i, j) \text{ for } \max(1, j-ku) \leq i \leq \min(n, j+kl).$ <p>The array <i>afb</i> contains details of the LU factorization of the banded matrix <i>A</i> as computed by <code>?gbtrf</code>. <i>U</i> is stored as an upper triangular banded matrix with <i>kl</i> + <i>ku</i> superdiagonals in rows 1 to <i>kl</i> + <i>ku</i> + 1. The multipliers used during the factorization are stored in rows <i>kl</i> + <i>ku</i> + 2 to $2*kl + ku + 1$.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p> <p><i>work</i>(*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; $ldab \geq kl + ku + 1$.
<i>ldafb</i>	INTEGER. The leading dimension of the array <i>afb</i> ; $ldafb \geq 2*kl + ku + 1$.
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains the pivot indices as computed by <code>?gbtrf</code>; for row $1 \leq i \leq n$, row <i>i</i> of the matrix was interchanged with row <i>ipiv</i>(<i>i</i>).</p>
<i>r, c</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Arrays: <i>r</i>(<i>n</i>), <i>c</i>(<i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>.</p> <p>If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag</i>(<i>r</i>); if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag</i>(<i>c</i>); if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p> <p>If <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.</p>

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.						
<i>x</i>	<p>REAL for <code>sgbrfsx</code></p> <p>DOUBLE PRECISION for <code>dgbrfsx</code></p> <p>COMPLEX for <code>cgbrfsx</code></p> <p>DOUBLE COMPLEX for <code>zgbrfsx</code>.</p> <p>Array, size (<i>ldx</i>, *).</p> <p>The solution matrix <i>X</i> as computed by <code>sgbtrs/dgbtrs</code> for real flavors or <code>cgbtrs/zgbtrs</code> for complex flavors.</p>						
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.						
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right-hand side and each type (normwise or componentwise). See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below.						
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.						
<i>params</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size <i>nparams</i>. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params</i>(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p> <table border="0" style="margin-left: 2em;"> <tr> <td style="padding-right: 2em;">=0.0</td> <td>No refinement is performed and no error bounds are computed.</td> </tr> <tr> <td style="padding-right: 2em;">=1.0</td> <td>Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.</td> </tr> </table> <p>(Other values are reserved for future use.)</p> <p><i>params</i>(2) : Maximum number of residual computations allowed for refinement.</p> <table border="0" style="margin-left: 2em;"> <tr> <td style="padding-right: 2em;">Default</td> <td>10.0</td> </tr> </table>	=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.	Default	10.0
=0.0	No refinement is performed and no error bounds are computed.						
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.						
Default	10.0						

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for *sgbrfsx*
 DOUBLE PRECISION for *dgbrfsx*
 COMPLEX for *cgbrfsx*
 DOUBLE COMPLEX for *zgbrfsx*.
 The improved solution matrix *X*.

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:
 Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i, :)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:, err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are
- $$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$
- Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`params`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in `params(1)`, `params(2)`, and `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < info \leq n$: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*(3) = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that $err_bnds_norm(j,1) = 0.0$ or $err_bnds_comp(j,1) = 0.0$. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal coefficient matrix and estimates its error.

Syntax

```
call sgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, iwork, info )
call dgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, iwork, info )
call cgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, rwork, info )
call zgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, rwork, info )
call gtrfs( dl, d, du, dlf, df, duf, du2, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ or $A^T * X = B$ or $A^H * X = B$ with a tridiagonal matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$$|\delta a_{ij}| / |a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| / |b_i| \leq \beta |b_i| \quad \text{such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?gttrf`
- call the solver routine `?gttrs`.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix <i>B</i> ; $nrhs \geq 0$.
<i>dl,d,du,dlf,</i> <i>df,duf,du2,</i> <i>b,x,work</i>	REAL for <i>sgtrfs</i> DOUBLE PRECISION for <i>dgtrfs</i> COMPLEX for <i>cgtrfs</i> DOUBLE COMPLEX for <i>zgtrfs</i> . Arrays: <i>dl</i> , dimension $(n - 1)$, contains the subdiagonal elements of <i>A</i> . <i>d</i> , dimension (n) , contains the diagonal elements of <i>A</i> . <i>du</i> , dimension $(n - 1)$, contains the superdiagonal elements of <i>A</i> . <i>dlf</i> , dimension $(n - 1)$, contains the $(n - 1)$ multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> as computed by <code>?gttrf</code> . <i>df</i> , dimension (n) , contains the n diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> . <i>duf</i> , dimension $(n - 1)$, contains the $(n - 1)$ elements of the first superdiagonal of <i>U</i> . <i>du2</i> , dimension $(n - 2)$, contains the $(n - 2)$ elements of the second superdiagonal of <i>U</i> . <i>b(ldb, nrhs)</i> contains the right-hand side matrix <i>B</i> . <i>x(ldx, nrhs)</i> contains the solution matrix <i>X</i> , as computed by <code>?gttrs</code> . <i>work(*)</i> is a workspace array; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by <code>?gttrf</code> .
<i>iwork</i>	INTEGER. Workspace array, size (n) . Used for real flavors only.

rwork REAL for `cgtrfs`
 DOUBLE PRECISION for `zgtrfs`.
 Workspace array, size (n). Used for complex flavors only.

Output Parameters

x The refined solution matrix X .

ferr, berr REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gtrfs` interface are as follows:

<i>dl</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>dlf</i>	Holds the vector of length $(n-1)$.
<i>df</i>	Holds the vector of length n .
<i>duf</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

See Also

Matrix Storage Schemes

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
call sporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call dporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork, info )
```

```
call cporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call zporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call porfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?potrf](#)
- call the solver routine [?potrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	REAL for <i>sporfs</i> DOUBLE PRECISION for <i>dporfs</i> COMPLEX for <i>cporfs</i> DOUBLE COMPLEX for <i>zporfs</i> . Arrays:

$a(lda, *)$ contains the original matrix A , as supplied to `?potrf`.
 $af(ldaf, *)$ contains the factored matrix A , as returned by `?potrf`.
 $b ldb, *)$ contains the right-hand side matrix B .
 $x(ldx, *)$ contains the solution matrix X .
 $work(*)$ is a workspace array.

The second dimension of a and af must be at least $\max(1, n)$; the second dimension of b and x must be at least $\max(1, nrhs)$; the dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of af ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>cporfs</code> DOUBLE PRECISION for <code>zporfs</code> . Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix X .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `porfs` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.

<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?porfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric/Hermitian positive-definite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call sporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call dporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call cporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, ldx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

If $uplo = 'L'$, the lower triangle of A is stored.

equed

CHARACTER*1. Must be 'N' or 'Y'.

Specifies the form of equilibration that was done to A before calling this routine.

If $equed = 'N'$, no equilibration was done.

If $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$. The right-hand side B has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

a, af, b, work

REAL for `sporfsx`

DOUBLE PRECISION for `dporfsx`

COMPLEX for `cporfsx`

DOUBLE COMPLEX for `zporfsx`.

Arrays: $a(lda, *)$, $af(ldaf, *)$, $b ldb, *)$, $work(*)$.

The array a contains the symmetric/Hermitian matrix A as specified by $uplo$. If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced. The second dimension of a must be at least $\max(1, n)$.

The array af contains the triangular factor L or U from the Cholesky factorization $A = U^*T^*U$ or $A = L^*L^*T$ as computed by `spotrf` for real flavors or `dpotrf` for complex flavors.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of af ; $ldaf \geq \max(1, n)$.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size n . The array s contains the scale factors for A .

If $equed = 'N'$, s is not accessed.

If *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>x</i>	REAL for <i>sporfsx</i> DOUBLE PRECISION for <i>dporfsx</i> COMPLEX for <i>cporfsx</i> DOUBLE COMPLEX for <i>zporfsx</i> . Array, size <i>ldx</i> by *. The solution matrix <i>X</i> as computed by ?potrs
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors). =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision. (Other values are reserved for future use.) <i>params</i> (2) : Maximum number of residual computations allowed for refinement. Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for *sporfsx*
 DOUBLE PRECISION for *dporfsx*
 COMPLEX for *cporfsx*
 DOUBLE COMPLEX for *zporfsx*.
 The improved solution matrix *X*.

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:
 Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix <i>Z</i> are $\ Z\ _{\infty} \cdot \ Z^{-1}\ _{\infty}$ Let $z = s * a$, where <i>s</i> scales each row by a power of the radix so all absolute row sums of <i>z</i> are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`params`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely in `params(1)`, `params(2)`, or `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info},\text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*(3) = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that $\text{err_bnds_norm}(j,1) = 0.0$ or $\text{err_bnds_comp}(j,1) = 0.0$. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?pprfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix stored in a packed format and estimates its error.

Syntax

```
call spprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call cpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call pprfs( ap, afp, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution

$$\|x - x_e\|_\infty / \|x\|_\infty$$

where x_e is the exact solution.

Before calling this routine:

- call the factorization routine [?pptrf](#)

- call the solver routine `?pptrs`.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates how the input matrix <i>A</i> has been factored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides; $nrhs \geq 0$.</p>
<i>ap, afp, b, x, work</i>	<p>REAL for <code>spprfs</code></p> <p>DOUBLE PRECISION for <code>dpprfs</code></p> <p>COMPLEX for <code>cpprfs</code></p> <p>DOUBLE COMPLEX for <code>zpprfs</code>.</p> <p>Arrays:</p> <p><i>ap</i>(*) contains the original matrix <i>A</i> in a packed format, as supplied to <code>?pptrf</code>.</p> <p><i>afp</i>(*) contains the factored matrix <i>A</i> in a packed format, as returned by <code>?pptrf</code>.</p> <p><i>b</i>(<i>ldb</i>,*) contains the right-hand side matrix <i>B</i>.</p> <p><i>x</i>(<i>ldx</i>,*) contains the solution matrix <i>X</i>.</p> <p><i>work</i>(*) is a workspace array.</p> <p>The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; $ldb \geq \max(1, n)$.</p>
<i>ldx</i>	<p>INTEGER. The leading dimension of <i>x</i>; $ldx \geq \max(1, n)$.</p>
<i>iwork</i>	<p>INTEGER. Workspace array, size at least $\max(1, n)$.</p>
<i>rwork</i>	<p>REAL for <code>cpprfs</code></p> <p>DOUBLE PRECISION for <code>zpprfs</code>.</p> <p>Workspace array, size at least $\max(1, n)$.</p>

Output Parameters

<i>x</i>	<p>The refined solution matrix <i>X</i>.</p>
<i>ferr, berr</i>	<p>REAL for single precision flavors.</p> <p>DOUBLE PRECISION for double precision flavors.</p>

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info*=0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pprfs` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1))/2$.
<i>afp</i>	Holds the array <i>AF</i> of size $(n*(n+1))/2$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
call spbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
```

```
call dpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
```

```
call cpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

```
call zpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

```
call pbrfs( ab, afb, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric (Hermitian) positive definite band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pbtrf](#)
- call the solver routine [?pbtrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for <code>spbrfs</code> DOUBLE PRECISION for <code>dpbrfs</code> COMPLEX for <code>cpbrfs</code> DOUBLE COMPLEX for <code>zpbrfs</code> . Arrays: <i>ab(ldab,*)</i> contains the original band matrix A , as supplied to ?pbtrf . <i>afb(ldafb,*)</i> contains the factored band matrix A , as returned by ?pbtrf . <i>b(ldb,*)</i> contains the right-hand side matrix B . <i>x(ldx,*)</i> contains the solution matrix X . <i>work(*)</i> is a workspace array.

The second dimension of ab and afb must be at least $\max(1, n)$; the second dimension of b and x must be at least $\max(1, nrhs)$; the dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>ldab</i>	INTEGER. The leading dimension of ab ; $ldab \geq kd + 1$.
<i>ldafb</i>	INTEGER. The leading dimension of afb ; $ldafb \geq kd + 1$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>cpbrfs</code> DOUBLE PRECISION for <code>zpbrfs</code> . Workspace array, size at least $\max(1, n)$.

Output Parameters

x	The refined solution matrix X .
<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbrfs` interface are as follows:

<i>ab</i>	Holds the array A of size $(kd+1, n)$.
<i>afb</i>	Holds the array AF of size $(kd+1, n)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n*kd$ floating-point operations (for real flavors) or $32n*kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n*kd$ operations (for real flavors) or $48n*kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n*kd$ floating-point operations for real flavors or $16n*kd$ for complex flavors.

See Also

Matrix Storage Schemes

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix and estimates its error.

Syntax

```
call sptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call dptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call zptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call ptrfs( d, df, e, ef, b, x [,ferr] [,berr] [,info] )
call ptrfs( d, df, e, ef, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pttrf](#)
- call the solver routine [?pttrs](#).

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Used for complex flavors only. Must be 'U' or 'L'.</p> <p>Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix <i>A</i> is stored and how <i>A</i> is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>e</i> stores the superdiagonal of <i>A</i>, and <i>A</i> is factored as $U^H * D * U$.</p> <p>If <i>uplo</i> = 'L', the array <i>e</i> stores the subdiagonal of <i>A</i>, and <i>A</i> is factored as $L * D * L^H$.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides; $nrhs \geq 0$.</p>
<i>d, df, rwork</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors</p> <p>Arrays: <i>d</i>(<i>n</i>), <i>df</i>(<i>n</i>), <i>rwork</i>(<i>n</i>).</p> <p>The array <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>A</i>.</p> <p>The array <i>df</i> contains the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization of <i>A</i> as computed by ?ptrf.</p> <p>The array <i>rwork</i> is a workspace array used for complex flavors only.</p>
<i>e,ef,b,x,work</i>	<p>REAL for sptrfs</p> <p>DOUBLE PRECISION for dptrfs</p> <p>COMPLEX for cptrfs</p> <p>DOUBLE COMPLEX for zptrfs.</p> <p>Arrays: <i>e</i>(<i>n</i> - 1), <i>ef</i>(<i>n</i> - 1), <i>b</i>(<i>ldb</i>, <i>nrhs</i>), <i>x</i>(<i>ldx</i>, <i>nrhs</i>), <i>work</i>(*).</p> <p>The array <i>e</i> contains the (<i>n</i> - 1) off-diagonal elements of the tridiagonal matrix <i>A</i> (see <i>uplo</i>).</p> <p>The array <i>ef</i> contains the (<i>n</i> - 1) off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by ?ptrf (see <i>uplo</i>).</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p> <p>The array <i>x</i> contains the solution matrix <i>X</i> as computed by ?ptrs.</p> <p>The array <i>work</i> is a workspace array. The dimension of <i>work</i> must be at least $2 * n$ for real flavors, and at least <i>n</i> for complex flavors.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; $ldb \geq \max(1, n)$.</p>
<i>ldx</i>	<p>INTEGER. The leading dimension of <i>x</i>; $ldx \geq \max(1, n)$.</p>

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
----------	--

ferr, berr REAL for single precision flavors.
 DOUBLE PRECISION for double precision flavors.
 Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ptrfs` interface are as follows:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>df</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>ef</i>	Holds the vector of length (<i>n</i> -1).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>uplo</i>	Used in complex flavors only. Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?syrfs

Refines the solution of a system of linear equations with a symmetric coefficient matrix and estimates its error.

Syntax

```
call ssyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
iwork, info )
```

```
call dsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
iwork, info )
```

```
call csyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

```
call zsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

```
call syrfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a symmetric full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?sytrf](#)
- call the solver routine [?sytrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	REAL for <code>ssyrfs</code> DOUBLE PRECISION for <code>dsyrfs</code> COMPLEX for <code>csyrfs</code> DOUBLE COMPLEX for <code>zsyrfs</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the original matrix A , as supplied to ?sytrf . <i>af</i> (<i>ldaf</i> ,*) contains the factored matrix A , as returned by ?sytrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> ,*) contains the solution matrix X . <i>work</i> (*) is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.

<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ? sytrf .
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>csyrfs</code> DOUBLE PRECISION for <code>zsyrfs</code> . Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `syrfs` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?syrfSX

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call ssyrfSX( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call dsyrfSX( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call csyrfSX( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zsyrfSX( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. Specifies the form of equilibration that was done to A before calling this routine. If <i>equed</i> = 'N', no equilibration was done.

If *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$. The right-hand side *B* has been changed accordingly.

n INTEGER. The number of linear equations; the order of the matrix *A*; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; the number of columns of the matrices *B* and *X*; $nrhs \geq 0$.

a, af, b, work REAL for `ssyrfsx`
 DOUBLE PRECISION for `dsyrfsx`
 COMPLEX for `csyrfsx`
 DOUBLE COMPLEX for `zsyrfxsx`.

Arrays: *a(lda,*)*, *af(ldaf,*)*, *b(ldb,*)*, *work(*)*.

The array *a* contains the symmetric/Hermitian matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least $\max(1, n)$.

The array *af* contains the triangular factor *L* or *U* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ as computed by `ssytrf` for real flavors or `dsytrf` for complex flavors.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work()* is a workspace array. The dimension of *work* must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of *D* as determined by `ssytrf` for real flavors or `dsytrf` for complex flavors.

s REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*.

If *equed* = 'N', *s* is not accessed.

If *equed* = 'Y', each element of *s* must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>x</i>	REAL for <code>ssyrfsx</code> DOUBLE PRECISION for <code>dsyrfsx</code> COMPLEX for <code>csyrfsx</code> DOUBLE COMPLEX for <code>zsyrfsx</code> . Array, size <i>ldx</i> by *. The solution matrix X as computed by <code>?sytrs</code>
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors). =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision. (Other values are reserved for future use.) <i>params</i> (2) : Maximum number of residual computations allowed for refinement. Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for *ssyrfsx*
 DOUBLE PRECISION for *dsyrfsx*
 COMPLEX for *csyrfsx*
 DOUBLE COMPLEX for *zsyrfsx*.
 The improved solution matrix *X*.

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:
 Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`params`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in `params(1)`, `params(2)`, `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

`info`

INTEGER. If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info},\text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j,1) = 0.0$ or $\text{err_bnds_comp}(j,1) = 0.0$. See the definition of err_bnds_norm and err_bnds_comp for $\text{err} = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?herfs

Refines the solution of a system of linear equations with a complex Hermitian coefficient matrix and estimates its error.

Syntax

```
call cherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call zherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call herfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a complex Hermitian full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hetrf](#)
- call the solver routine [?hetrs](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', the upper triangle of A is stored.

If `uplo = 'L'`, the lower triangle of A is stored.

`n` INTEGER. The order of the matrix A ; $n \geq 0$.

`nrhs` INTEGER. The number of right-hand sides; $nrhs \geq 0$.

`a,af,b,x,work` COMPLEX for `cherfs`
DOUBLE COMPLEX for `zherfs`.

Arrays:

`a` (size `lda` by `*`) contains the original matrix A , as supplied to `?hetrf`.

`af` (size `ldaf` by `*`) contains the factored matrix A , as returned by `?hetrf`.

`b` (size `ldb` by `*`) contains the right-hand side matrix B .

`x` (size `ldx` by `*`) contains the solution matrix X .

`work`(`*`) is a workspace array.

The second dimension of `a` and `af` must be at least $\max(1, n)$; the second dimension of `b` and `x` must be at least $\max(1, nrhs)$; the dimension of `work` must be at least $\max(1, 2*n)$.

`lda` INTEGER. The leading dimension of `a`; $lda \geq \max(1, n)$.

`ldaf` INTEGER. The leading dimension of `af`; $ldaf \geq \max(1, n)$.

`ldb` INTEGER. The leading dimension of `b`; $ldb \geq \max(1, n)$.

`ldx` INTEGER. The leading dimension of `x`; $ldx \geq \max(1, n)$.

`ipiv` INTEGER.
Array, size at least $\max(1, n)$. The `ipiv` array, as returned by `?hetrf`.

`rwork` REAL for `cherfs`
DOUBLE PRECISION for `zherfs`.
Workspace array, size at least $\max(1, n)$.

Output Parameters

`x` The refined solution matrix X .

`ferr, berr` REAL for `cherfs`
DOUBLE PRECISION for `zherfs`.

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

`info` INTEGER. If `info = 0`, the execution is successful.
If `info = -i`, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `herfs` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>af</code>	Holds the matrix AF of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>x</code>	Holds the matrix X of size $(n, nrhs)$.
<code>ferr</code>	Holds the vector of length $(nrhs)$.
<code>berr</code>	Holds the vector of length $(nrhs)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssyrfs/?dsyrfs](#)

See Also

[Matrix Storage Schemes](#)

[?herfsx](#)

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
call cherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed` and `s` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If <code>uplo</code> = 'U', the upper triangle of A is stored.</p> <p>If <code>uplo</code> = 'L', the lower triangle of A is stored.</p>
<code>equed</code>	<p>CHARACTER*1. Must be 'N' or 'Y'.</p> <p>Specifies the form of equilibration that was done to A before calling this routine.</p> <p>If <code>equed</code> = 'N', no equilibration was done.</p> <p>If <code>equed</code> = 'Y', both row and column equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$. The right-hand side B has been changed accordingly.</p>
<code>n</code>	<p>INTEGER. The number of linear equations; the order of the matrix A; $n \geq 0$.</p>
<code>nrhs</code>	<p>INTEGER. The number of right-hand sides; the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<code>a, af, b, work</code>	<p>COMPLEX for <code>cherfsx</code></p> <p>DOUBLE COMPLEX for <code>zherfsx</code>.</p> <p>Arrays: <code>a(lda,*)</code>, <code>af(ldaf,*)</code>, <code>b ldb,*)</code>, <code>work(*)</code>.</p> <p>The array <code>a</code> contains the Hermitian matrix A as specified by <code>uplo</code>. If <code>uplo</code> = 'U', the leading n-by-n upper triangular part of <code>a</code> contains the upper triangular part of the matrix A and the strictly lower triangular part of <code>a</code> is not referenced. If <code>uplo</code> = 'L', the leading n-by-n lower triangular part of <code>a</code> contains the lower triangular part of the matrix A and the strictly upper triangular part of <code>a</code> is not referenced. The second dimension of <code>a</code> must be at least $\max(1, n)$.</p> <p>The array <code>af</code> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U * D * U^T$ or $A = L * D * L^T$ as computed by <code>ssytrf</code> for <code>cherfsx</code> or <code>dsytrf</code> for <code>zherfsx</code>.</p> <p>The array <code>b</code> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <code>b</code> must be at least $\max(1, nrhs)$.</p> <p><code>work(*)</code> is a workspace array. The dimension of <code>work</code> must be at least $\max(1, 2 * n)$.</p>
<code>lda</code>	<p>INTEGER. The leading dimension of <code>a</code>; $lda \geq \max(1, n)$.</p>

<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> as determined by ssytrf for real flavors or dsytrf for complex flavors.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i> . If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>equed</i> = 'Y', each element of <i>s</i> must be positive. Each element of <i>s</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>x</i>	COMPLEX for cherfsx DOUBLE COMPLEX for zherfsx . Array, size (<i>ldx</i> , *). The solution matrix <i>X</i> as computed by ?hetrs
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for cherfsx), 1.0D+0 (for zherfsx). =0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default 10

Aggressive Set to 100 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

rwork

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Workspace array, size at least $\max(1, 3*n)$.

Output Parameters

x

COMPLEX for *cherfsx*

DOUBLE COMPLEX for *zherfsx*.

The improved solution matrix *X*.

rcond

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i, :)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:, err)* contains the following three fields:

- err*=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for *cherfsx* and $\sqrt{n} * \text{dlamch}(\epsilon)$ for *zherfsx*.
- err*=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for *cherfsx* and $\sqrt{n} * \text{dlamch}(\epsilon)$ for *zherfsx*. This error bound should only be trusted if the previous boolean is true.
- err*=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

err_bnds_comp

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for `cherfsx` and $\sqrt{n} * dlamch(\epsilon)$ for `zherfsx`.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for `cherfsx` and $\sqrt{n} * dlamch(\epsilon)$ for `zherfsx`. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`params`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in `params(1)`, `params(2)`, `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < info \leq n$: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*(3) = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that $err_bnds_norm(j,1) = 0.0$ or $err_bnds_comp(j,1) = 0.0$. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?sprfs

Refines the solution of a system of linear equations with a packed symmetric coefficient matrix and estimates the solution error.

Syntax

```
call ssprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork,
info )
call dsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork,
info )
call csprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call zsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call sprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a packed symmetric matrix *A*, with multiple right-hand sides. For each computed solution vector *x*, the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of *A* and *b* such that *x* is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?spturf](#)
- call the solver routine [?sptrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	REAL for <i>ssprfs</i> DOUBLE PRECISION for <i>dsprfs</i> COMPLEX for <i>csprfs</i> DOUBLE COMPLEX for <i>zsprfs</i> . Arrays: <i>ap</i> (size *) contains the original packed matrix <i>A</i> , as supplied to ?spturf . <i>afp</i> (size *) contains the factored packed matrix <i>A</i> , as returned by ?spturf . <i>b</i> (size <i>ldb</i> by *) contains the right-hand side matrix <i>B</i> . <i>x</i> (size <i>ldx</i> by *) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?spturf .
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>csprfs</i> DOUBLE PRECISION for <i>zsprfs</i> . Workspace array, size at least $\max(1, n)$.

Output Parameters

x The refined solution matrix *X*.

<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sprfs` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>afp</i>	Holds the array <i>AF</i> of size $(n*(n+1)/2)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian coefficient matrix and estimates the solution error.

Syntax

```
call chprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call zhprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

```
call hprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a packed complex Hermitian matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hptrf](#)
- call the solver routine [?hptrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	COMPLEX for <code>chprfs</code> DOUBLE COMPLEX for <code>zhprfs</code> . Arrays: <i>ap</i> (size *) contains the original packed matrix A , as supplied to ?hptrf . <i>afp</i> (size *) contains the factored packed matrix A , as returned by ?hptrf . <i>b</i> (size <i>ldb</i> by *) contains the right-hand side matrix B . <i>x</i> (size <i>ldx</i> by *) contains the solution matrix X . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.

<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ? hptrf .
<i>rwork</i>	REAL for <i>chprfs</i> DOUBLE PRECISION for <i>zhprfs</i> . Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for <i>chprfs</i> . DOUBLE PRECISION for <i>zhprfs</i> . Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hprfs* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>afp</i>	Holds the array <i>AF</i> of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssprfs/?dsprfs](#).

See Also

Matrix Storage Schemes

[?trrfs](#)

Estimates the error in the solution of a system of linear equations with a triangular coefficient matrix.

Syntax

```
call strrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
```

```
call dtrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
            iwork, info )
```

```
call ctrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call ztrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
            rwork, info )
```

```
call trrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?trtrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', then *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

n INTEGER. The order of the matrix *A*; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a, b, x, work REAL for *strrfs*
DOUBLE PRECISION for *dtrrfs*
COMPLEX for *ctrfrs*
DOUBLE COMPLEX for *ztrrfs*.

Arrays:

a (size *lda* by *) contains the upper or lower triangular matrix *A*, as specified by *uplo*.

b (size *ldb* by *) contains the right-hand side matrix *B*.

x (size *ldx* by *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *a* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

iwork INTEGER. Workspace array, size at least $\max(1, n)$.

rwork REAL for *ctrfrs*
DOUBLE PRECISION for *ztrrfs*.
Workspace array, size at least $\max(1, n)$.

Output Parameters

ferr, berr REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trrfs` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>x</code>	Holds the matrix X of size $(n, nrhs)$.
<code>ferr</code>	Holds the vector of length $(nrhs)$.
<code>berr</code>	Holds the vector of length $(nrhs)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<code>diag</code>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?tprfs

Estimates the error in the solution of a system of linear equations with a packed triangular coefficient matrix.

Syntax

```
call stprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dtprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call ctprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call ztprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call tprfs( ap, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a packed triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tprfs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, b, x, work</i>	REAL for <i>stprfs</i> DOUBLE PRECISION for <i>dtprfs</i> COMPLEX for <i>ctprfs</i> DOUBLE COMPLEX for <i>ztprfs</i> . Arrays: <i>ap</i> (size *) contains the upper or lower triangular matrix A , as specified by <i>uplo</i> . <i>b</i> (size <i>ldb</i> by *) contains the right-hand side matrix B . <i>x</i> (size <i>ldx</i> by *) contains the solution matrix X . <i>work</i> (*) is a workspace array.

The dimension of ap must be at least $\max(1, n(n+1)/2)$; the second dimension of b and x must be at least $\max(1, nrhs)$; the dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>ctprfs</code> DOUBLE PRECISION for <code>ztpfrfs</code> . Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tprfs` interface are as follows:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band coefficient matrix.

Syntax

```
call stbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
```

```
call dtbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
```

```
call ctbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

```
call ztbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

```
call tbrfs( ab, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tbtrs](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$.

If $trans = 'C'$, the system has the form $A^H * X = B$.

diag

CHARACTER*1. Must be 'N' or 'U'.

If $diag = 'N'$, A is not a unit triangular matrix.

If $diag = 'U'$, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array ab .

n

INTEGER. The order of the matrix A ; $n \geq 0$.

kd

INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ; $kd \geq 0$.

nrhs

INTEGER. The number of right-hand sides; $nrhs \geq 0$.

ab, b, x, work

REAL for *stbrfs*

DOUBLE PRECISION for *dtbrfs*

COMPLEX for *ctbrfs*

DOUBLE COMPLEX for *ztbrfs*.

Arrays:

ab(size *ldab* by *) contains the upper or lower triangular matrix A , as specified by *uplo*, in band storage format.

b(size *ldb* by *) contains the right-hand side matrix B .

x(size *ldx* by *) contains the solution matrix X .

work(*) is a workspace array.

The second dimension of a must be at least $\max(1, n)$; the second dimension of b and x must be at least $\max(1, nrhs)$. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldab

INTEGER. The leading dimension of the array ab ; $ldab \geq kd + 1$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.

iwork

INTEGER. Workspace array, size at least $\max(1, n)$.

rwork

REAL for *ctbrfs*

DOUBLE PRECISION for *ztbrfs*.

Workspace array, size at least $\max(1, n)$.

Output Parameters

ferr, berr

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tbrfs` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

See Also

[Matrix Storage Schemes](#)

Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Routines for Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

```
call sgetri( n, a, lda, ipiv, work, lwork, info )
call dgetri( n, a, lda, ipiv, work, lwork, info )
call cgetri( n, a, lda, ipiv, work, lwork, info )
```

call `zgetri(n, a, lda, ipiv, work, lwork, info)`

call `getri(a, ipiv [,info])`

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a general matrix A . Before calling this routine, call `?getrf` to factorize A .

Input Parameters

<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.
<code>a, work</code>	REAL for <code>sgetri</code> DOUBLE PRECISION for <code>dgetri</code> COMPLEX for <code>cgetri</code> DOUBLE COMPLEX for <code>zgetri</code> . Arrays: <code>a(lda,*)</code> , <code>work(*)</code> . <code>a(lda,*)</code> contains the factorization of the matrix A , as returned by <code>?getrf</code> : $A = P*L*U$. The second dimension of <code>a</code> must be at least $\max(1, n)$. <code>work(*)</code> is a workspace array of dimension at least $\max(1, lwork)$.
<code>lda</code>	INTEGER. The leading dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>ipiv</code>	INTEGER. Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by <code>?getrf</code> .
<code>lwork</code>	INTEGER. The size of the <code>work</code> array; $lwork \geq n$. If <code>lwork = -1</code> , then a workspace query is assumed; the routine only calculates the optimal size of the <code>work</code> array, returns this value as the first entry of the <code>work</code> array, and no error message related to <code>lwork</code> is issued by <code>xerbla</code> .

See *Application Notes* below for the suggested value of `lwork`.

Output Parameters

<code>a</code>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of the factor U is zero, U is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `getri` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed inverse X satisfies the following error bound:

$$\|XA - I\| \leq c(n)\epsilon \|X\| \|P\| \|L\| \|U\|,$$

where $c(n)$ is a modest linear function of n ; ϵ is the machine precision; I denotes the identity matrix; P , L , and U are the factors of the matrix factorization $A = P^*L^*U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Syntax

```
call spotri( uplo, n, a, lda, info )
call dpotri( uplo, n, a, lda, info )
call cpotri( uplo, n, a, lda, info )
call zpotri( uplo, n, a, lda, info )
call potri( a [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A . Before calling this routine, call [?potrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for <code>spotri</code> DOUBLE PRECISION for <code>dpotri</code> COMPLEX for <code>cpotri</code> DOUBLE COMPLEX for <code>zpotri</code> . Array <i>a</i> (size <i>lda</i> by *). Contains the factorization of the matrix A , as returned by ?potrf . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the upper or lower triangle of the inverse of A .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value. If <i>info</i> = i , the i -th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `potri` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n) \varepsilon \kappa_2(A), \quad \|AX - I\|_2 \leq c(n) \varepsilon \kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix A is defined by $\|A\|_2 = \max_{x: \|x\|=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?pftri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix in RFP format using the Cholesky factorization.

Syntax

```
call spftri( transr, uplo, n, a, info )
call dpftri( transr, uplo, n, a, info )
call cpftri( transr, uplo, n, a, info )
call zpftri( transr, uplo, n, a, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex data, Hermitian positive-definite matrix A using the Cholesky factorization:

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

Before calling this routine, call [?pfftrf](#) to factorize A .

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

transr CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).
 If *transr* = 'N', the Normal *transr* of RFP U (if *uplo* = 'U') or L (if *uplo* = 'L') is stored.
 If *transr* = 'T', the Transpose *transr* of RFP U (if *uplo* = 'U') or L (if *uplo* = 'L') is stored.
 If *transr* = 'C', the Conjugate-Transpose *transr* of RFP U (if *uplo* = 'U') or L (if *uplo* = 'L') is stored.

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates how the input matrix <i>A</i> has been factored:</p> <p>If <i>uplo</i> = 'U', $A = U^T * U$ for real data or $A = U^H * U$ for complex data, and <i>U</i> is stored.</p> <p>If <i>uplo</i> = 'L', $A = L * L^T$ for real data or $A = L * L^H$ for complex data, and <i>L</i> is stored.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	<p>REAL for <code>spptri</code></p> <p>DOUBLE PRECISION for <code>dpptri</code></p> <p>COMPLEX for <code>cpptri</code></p> <p>DOUBLE COMPLEX for <code>zpptri</code>.</p> <p>Array, size $(n * (n + 1) / 2)$. The array <i>a</i> contains the factor <i>U</i> or <i>L</i> matrix <i>A</i> in the RFP format.</p>

Output Parameters

<i>a</i>	The symmetric/Hermitian inverse of the original matrix in the same storage format.
<i>info</i>	<p>INTEGER. If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, the (<i>i</i>, <i>i</i>) element of the factor <i>U</i> or <i>L</i> is zero, and the inverse could not be computed.</p>

See Also

Matrix Storage Schemes

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix using Cholesky factorization.

Syntax

```
call spptri( uplo, n, ap, info )
call dpptri( uplo, n, ap, info )
call cpptri( uplo, n, ap, info )
call zpptri( uplo, n, ap, info )
call pptri( ap [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix *A* in *packed* form. Before calling this routine, call `?pptrf` to factorize *A*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular factor is stored in <i>ap</i> : If <i>uplo</i> = 'U', then the upper triangular factor is stored. If <i>uplo</i> = 'L', then the lower triangular factor is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>ap</i>	REAL for <i>sptrf</i> DOUBLE PRECISION for <i>dptrf</i> COMPLEX for <i>cptrf</i> DOUBLE COMPLEX for <i>zptrf</i> . Array, size at least $\max(1, n(n+1)/2)$. Contains the factorization of the packed matrix <i>A</i> , as returned by ?ptrf . The dimension <i>ap</i> must be at least $\max(1, n(n+1)/2)$.

Output Parameters

<i>ap</i>	Overwritten by the packed <i>n</i> -by- <i>n</i> matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *pptrf* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n) \varepsilon \kappa_2(A), \quad \|AX - I\|_2 \leq c(n) \varepsilon \kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix *A* is defined by $\|A\|_2 = \max_{x \cdot x = 1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?sytri

*Computes the inverse of a symmetric matrix using $U^*D^*U^T$ or $L^*D^*L^T$ Bunch-Kaufman factorization.*

Syntax

```
call ssytri( uplo, n, a, lda, ipiv, work, info )
call dsytri( uplo, n, a, lda, ipiv, work, info )
call csytri( uplo, n, a, lda, ipiv, work, info )
call zsytri( uplo, n, a, lda, ipiv, work, info )
call sytri( a, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the Bunch-Kaufman factorization $A = U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the Bunch-Kaufman factorization $A = L^*D^*L^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for ssytri DOUBLE PRECISION for dsytri COMPLEX for csytri DOUBLE COMPLEX for zsytri. Arrays: <i>a</i> (size <i>lda</i> by *) contains the factorization of the matrix A , as returned by ?sytrf . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.

ipiv INTEGER.
 Array, size at least $\max(1, n)$.
 The *ipiv* array, as returned by `?sytrf`.

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of D is zero, D is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytri` interface are as follows:

a Holds the matrix A of size (n, n) .

ipiv Holds the vector of length n .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D^*U^T*P^T*X*P*U - I| \leq c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \leq c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?sytri_rook

Computes the inverse of a symmetric matrix using $U*D*U^T$ or $L*D*L^T$ bounded Bunch-Kaufman factorization.

Syntax

```
call ssytri_rook( uplo, n, a, lda, ipiv, work, info )
call dsytri_rook( uplo, n, a, lda, ipiv, work, info )
call csytri_rook( uplo, n, a, lda, ipiv, work, info )
```

```
call zsytri_rook( uplo, n, a, lda, ipiv, work, info )
```

```
call sytri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf_rook](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array a stores the factorization $A = U*D*U^T$. If <i>uplo</i> = 'L', the array a stores the factorization $A = L*D*L^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for <code>ssytri_rook</code> DOUBLE PRECISION for <code>dsytri_rook</code> COMPLEX for <code>csytri_rook</code> DOUBLE COMPLEX for <code>zsytri_rook</code> . Arrays: <i>a</i> (size <i>lda</i> by *) contains the factorization of the matrix A , as returned by ?sytrf_rook . The second dimension of a must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf_rook .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$. If <i>uplo</i> = 'U', the upper triangular part of the inverse is formed and the part of a below the diagonal is not referenced; if <i>uplo</i> = 'L' the lower triangular part of the inverse is formed and the part of a above the diagonal is not referenced."
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - i , the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytri_rook` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hetri

*Computes the inverse of a complex Hermitian matrix using $U^*D^*U^H$ or $L^*D^*L^H$ Bunch-Kaufman factorization.*

Syntax

```
call chetri( uplo, n, a, lda, ipiv, work, info )
call zhetri( uplo, n, a, lda, ipiv, work, info )
call hetri( a, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $inv(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf](#) to factorize A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the Bunch-Kaufman factorization $A = U^*D^*U^H$. If <code>uplo = 'L'</code> , the array <code>a</code> stores the Bunch-Kaufman factorization $A = L^*D^*L^H$.
<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.

a, *work* COMPLEX for `chetri`
 DOUBLE COMPLEX for `zhetri`.

Arrays:

a(*lda*,*) contains the factorization of the matrix *A*, as returned by [?hetrf](#).

The second dimension of *a* must be at least $\max(1, n)$.

work(*) is a workspace array.

The dimension of *work* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. The *ipiv* array, as returned by [?hetrf](#).

Output Parameters

a Overwritten by the *n*-by-*n* matrix $\text{inv}(A)$.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetri` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

ipiv Holds the vector of length *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D^*U^H * P^T * X * P * U - I| \leq c(n) \varepsilon (|D| |U^H| P^T |X| P |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|D^*L^H * P^T * X * P * L - I| \leq c(n) \varepsilon (|D| |L^H| P^T |X| P |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here *c*(*n*) is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is [?sytri](#).

See Also

Matrix Storage Schemes

[?hetri_rook](#)

Computes the inverse of a complex Hermitian matrix using $U^*D^*U^H$ or $L^*D^*L^H$ bounded Bunch-Kaufman factorization.

Syntax

```
call chetri_rook( uplo, n, a, lda, ipiv, work, info )
```

```
call zhetri_rook( uplo, n, a, lda, ipiv, work, info )
```

```
call hetri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf_rook](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	COMPLEX for <code>chetri_rook</code> DOUBLE COMPLEX for <code>zhetri_rook</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the factorization of the matrix A , as returned by ?hetrf_rook . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf_rook .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If $\text{info} = 0$, the execution is successful. If $\text{info} = -i$, the i -th parameter had an illegal value. If $\text{info} = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetri_rook` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is `?sytri_rook`.

See Also

[Matrix Storage Schemes](#)

?sytri2

Computes the inverse of a symmetric indefinite matrix through setting the leading dimension of the workspace and calling ?sytri2x.

Syntax

```
call ssytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call csytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call sytri2( a, ipiv[,uplo][,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix A using the factorization $A = U * D * U^T$ or $A = L * D * L^T$ computed by `?sytrf`.

The `?sytri2` routine sets the leading dimension of the workspace before calling `?sytri2x` that actually computes the inverse.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates how the input matrix <i>A</i> has been factored:</p> <p>If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U^*D^*U^T$.</p> <p>If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L^*D^*L^T$.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>; $n \geq 0$.</p>
<i>a, work</i>	<p>REAL for <i>ssytri2</i></p> <p>DOUBLE PRECISION for <i>dsytri2</i></p> <p>COMPLEX for <i>csytri2</i></p> <p>DOUBLE COMPLEX for <i>zsytri2</i></p> <p>Array <i>a</i> (size <i>lda</i> by <i>n</i>) contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as returned by <i>?sytrf</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array of $(n+nb+1) * (nb+3)$ dimension.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of <i>a</i>; $lda \geq \max(1, n)$.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$.</p> <p>Details of the interchanges and the block structure of <i>D</i> as returned by <i>?sytrf</i>.</p>
<i>lwork</i>	<p>INTEGER. The dimension of the <i>work</i> array.</p> <p>$lwork \geq (n+nb+1) * (nb+3)$</p> <p>where</p> <p><i>nb</i> is the block size parameter as returned by <i>sytrf</i>.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i>.</p>

Output Parameters

<i>a</i>	<p>If <i>info</i> = 0, the symmetric inverse of the original matrix.</p> <p>If <i>uplo</i> = 'U', the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced.</p> <p>If <i>uplo</i> = 'L', the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

If $info = i$, $D(i,i) = 0$; D is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytri2` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Indicates how the matrix A has been factored. Must be 'U' or 'L'.

See Also

[?sytrf](#)

[?sytri2x](#)

[Matrix Storage Schemes](#)

?hetri2

Computes the inverse of a Hermitian indefinite matrix through setting the leading dimension of the workspace and calling ?hetri2x.

Syntax

```
call chetri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetri2( uplo, n, a, lda, ipiv, work, lwork, info )
call hetri2( a, ipiv[,uplo][,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $inv(A)$ of a Hermitian indefinite matrix A using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by `?hetrf`.

The `?hetri2` routine sets the leading dimension of the workspace before calling `?hetri2x` that actually computes the inverse.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the factorization $A = U*D*U^H$. If <code>uplo = 'L'</code> , the array <code>a</code> stores the factorization $A = L*D*L^H$.
<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.
<code>a, work</code>	COMPLEX for <code>chetri2</code>

DOUBLE COMPLEX for `zhetri2`

Array `a` (size `lda` by `*`) contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as returned by `?sytrf`.

The second dimension of `a` must be at least $\max(1, n)$.

`work` is a workspace array of $(n+nb+1) * (nb+3)$ dimension.

`lda`

INTEGER. The leading dimension of `a`; $lda \geq \max(1, n)$.

`ipiv`

INTEGER.

Array, size at least $\max(1, n)$.

Details of the interchanges and the block structure of D as returned by `?hetrf`.

`lwork`

INTEGER. The dimension of the `work` array.

$lwork \geq (n+nb+1) * (nb+3)$

where

`nb` is the block size parameter as returned by `hetrf`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

Output Parameters

`a`

If `info = 0`, the inverse of the original matrix.

If `uplo = 'U'`, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.

If `uplo = 'L'`, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, $D(i,i) = 0$; D is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetri2` interface are as follows:

`a`

Holds the matrix A of size (n, n) .

`ipiv`

Holds the vector of length n .

uplo Indicates how the input matrix A has been factored. Must be 'U' or 'L'.

See Also

[?hetrf](#)

[?hetri2x](#)

Matrix Storage Schemes

[?sytri2x](#)

Computes the inverse of a symmetric indefinite matrix after [?sytri2](#) sets the leading dimension of the workspace.

Syntax

```
call ssytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call dsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call csytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call sytri2x( a, ipiv, nb[, uplo][, info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix A using the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ computed by [?sytrf](#).

The [?sytri2x](#) actually computes the inverse after the [?sytri2](#) routine sets the leading dimension of the workspace before calling [?sytri2x](#).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If *uplo* = 'U', the array *a* stores the factorization $A = U^*D^*U^T$.
If *uplo* = 'L', the array *a* stores the factorization $A = L^*D^*L^T$.

n INTEGER. The order of the matrix A ; $n \geq 0$.

a, work REAL for [ssytri2x](#)
DOUBLE PRECISION for [dsytri2x](#)
COMPLEX for [csytri2x](#)
DOUBLE COMPLEX for [zsytri2x](#)

Arrays:
a(lda,)* contains the *nb* (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by [?sytrf](#).
The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array of dimension $(n+nb+1) * (nb+3)$

where

nb is the block size as set by `?sytrf`.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ipiv

INTEGER.

Array, size at least $\max(1, n)$.

Details of the interchanges and the *nb* structure of *D* as returned by `?sytrf`.

nb

INTEGER. Block size.

Output Parameters

a

If *info* = 0, the symmetric inverse of the original matrix.

If *info* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, $D_{ii} = 0$; *D* is singular and its inversion could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytri2x` interface are as follows:

a

Holds the matrix *A* of size (n, n) .

ipiv

Holds the vector of length *n*.

nb

Holds the block size.

uplo

Indicates how the input matrix *A* has been factored. Must be 'U' or 'L'.

See Also

[?sytrf](#)

[?sytri2](#)

[Matrix Storage Schemes](#)

?hetri2x

Computes the inverse of a Hermitian indefinite matrix after ?hetri2 sets the leading dimension of the workspace.

Syntax

```
call chetri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zhetri2x( uplo, n, a, lda, ipiv, work, nb, info )
call hetri2x( a, ipiv, nb[, uplo][, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by ?hetrf.

The ?hetri2x actually computes the inverse after the ?hetri2 routine sets the leading dimension of the workspace before calling ?hetri2x.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for chetri2x DOUBLE COMPLEX for zhetri2x Arrays: <i>a</i> (<i>lda</i> ,*) contains the <i>nb</i> (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?hetrf. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array of the dimension $(n+nb+1) * (nb+3)$ where <i>nb</i> is the block size as set by ?hetrf.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Details of the interchanges and the <i>nb</i> structure of D as returned by ?hetrf.
<i>nb</i>	INTEGER. Block size.

Output Parameters

<i>a</i>	<p>If <i>info</i> = 0, the symmetric inverse of the original matrix.</p> <p>If <i>info</i> = 'U', the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced.</p> <p>If <i>info</i> = 'L', the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, $D_{ii} = 0$; <i>D</i> is singular and its inversion could not be computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetri2x` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>nb</i>	Holds the block size.
<i>uplo</i>	Indicates how the input matrix <i>A</i> has been factored. Must be 'U' or 'L'.

See Also

[?hetrf](#)

[?hetri2](#)

[Matrix Storage Schemes](#)

[?sptri](#)

*Computes the inverse of a symmetric matrix using $U^*D^*U^T$ or $L^*D^*L^T$ Bunch-Kaufman factorization of matrix in packed storage.*

Syntax

```
call ssptri( uplo, n, ap, ipiv, work, info )
call dsptri( uplo, n, ap, ipiv, work, info )
call csptri( uplo, n, ap, ipiv, work, info )
call zsptri( uplo, n, ap, ipiv, work, info )
call sptri( ap, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a packed symmetric matrix A . Before calling this routine, call [?sptrf](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = L^*D^*L^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap, work</i>	REAL for <i>ssptri</i> DOUBLE PRECISION for <i>dsptri</i> COMPLEX for <i>csptri</i> DOUBLE COMPLEX for <i>zsptri</i> . Arrays: <i>ap</i> (*) contains the factorization of the matrix A , as returned by ?sptrf . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sptrf .

Output Parameters

<i>ap</i>	Overwritten by the matrix $\text{inv}(A)$ in packed form.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *sptri* interface are as follows:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \leq c(n)\epsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for `uplo = 'U'`, and

$$|D*L^T*P^T*X*P*L - I| \leq c(n)\epsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for `uplo = 'L'`. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?hptri

*Computes the inverse of a complex Hermitian matrix using $U*D*U^H$ or $L*D*L^H$ Bunch-Kaufman factorization of matrix in packed storage.*

Syntax

```
call chptri( uplo, n, ap, ipiv, work, info )
call zhptri( uplo, n, ap, ipiv, work, info )
call hptri( ap, ipiv [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A using packed storage. Before calling this routine, call `?hptrf` to factorize A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array <code>ap</code> stores the packed Bunch-Kaufman factorization $A = U*D*U^H$. If <code>uplo = 'L'</code> , the array <code>ap</code> stores the packed Bunch-Kaufman factorization $A = L*D*L^H$.
<code>n</code>	INTEGER. The order of the matrix A ; $n \geq 0$.
<code>ap, work</code>	COMPLEX for <code>chptri</code>

DOUBLE COMPLEX for `zhptri`.

Arrays:

`ap(*)` contains the factorization of the matrix A , as returned by [?hptrf](#).

The dimension of `ap` must be at least $\max(1, n(n+1)/2)$.

`work(*)` is a workspace array.

The dimension of `work` must be at least $\max(1, n)$.

`ipiv`

INTEGER.

Array, size at least $\max(1, n)$.

The `ipiv` array, as returned by [?hptrf](#).

Output Parameters

`ap`

Overwritten by the matrix $\text{inv}(A)$.

`info`

INTEGER.

If `info` = 0, the execution is successful.

If `info` = $-i$, the i -th parameter had an illegal value.

If `info` = i , the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hptri` interface are as follows:

`ap`

Holds the array A of size $(n*(n+1)/2)$.

`ipiv`

Holds the vector of length n .

`uplo`

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \leq c(n)\epsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for `uplo` = 'U', and

$$|D*L^H*P^T*X*P*L - I| \leq c(n)\epsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for `uplo` = 'L'. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3)n^3$.

The real counterpart of this routine is [?sptri](#).

See Also

[Matrix Storage Schemes](#)

?trtri

Computes the inverse of a triangular matrix.

Syntax

```
call strtri( uplo, diag, n, a, lda, info )
call dtrtri( uplo, diag, n, a, lda, info )
call ctrtri( uplo, diag, n, a, lda, info )
call ztrtri( uplo, diag, n, a, lda, info )
call trtri( a [,uplo] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a triangular matrix A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for strtri DOUBLE PRECISION for dtrtri COMPLEX for ctrtri DOUBLE COMPLEX for ztrtri. Array: size <i>lda</i> by *size $\max(1, \text{lda} * n)$. Contains the matrix A . The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The first dimension of a ; $\text{lda} \geq \max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trtri` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>diag</code>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\| \leq c(n)\varepsilon \|X\| \|A\|$$

$$\|XA - I\| \leq c(n)\varepsilon \|A^{-1}\| \|A\| \|X\|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?tftri

Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format.

Syntax

```
call stftri( transr, uplo, diag, n, a, info )
call dtftri( transr, uplo, diag, n, a, info )
call ctftri( transr, uplo, diag, n, a, info )
call ztftri( transr, uplo, diag, n, a, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

Computes the inverse of a triangular matrix A stored in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

`transr` CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).

If *transr* = 'N', the Normal *transr* of RFP *A* is stored.
 If *transr* = 'T', the Transpose *transr* of RFP *A* is stored.
 If *transr* = 'C', the Conjugate-Transpose *transr* of RFP *A* is stored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of RFP *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.

diag

CHARACTER*1. Must be 'N' or 'U'.

If *diag* = 'N', then *A* is not a unit triangular matrix.

If *diag* = 'U', *A* is unit triangular: diagonal elements of *A* are assumed to be 1 and not referenced in the array *a*.

*n*INTEGER. The order of the matrix *A*; $n \geq 0$.*a*REAL for *stftri*DOUBLE PRECISION for *dtftri*COMPLEX for *ctftri*DOUBLE COMPLEX for *ztftri*.

Array, size $\max(1, n*(n + 1)/2)$. The array *a* contains the matrix *A* in the RFP format.

Output Parameters

a

The (triangular) inverse of the original matrix in the same storage format.

info

INTEGER. If *info*=0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

If *info* = *i*, $A(i, i)$ is exactly zero. The triangular matrix is singular and its inverse cannot be computed.

See Also

Matrix Storage Schemes

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

```
call stptri( uplo, diag, n, ap, info )
```

```
call dtptri( uplo, diag, n, ap, info )
```

```
call ctptri( uplo, diag, n, ap, info )
```

```
call ztptri( uplo, diag, n, ap, info )
call tptri( ap [,uplo] [,diag] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the inverse $\text{inv}(A)$ of a packed triangular matrix A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap</i>	REAL for <i>stptri</i> DOUBLE PRECISION for <i>dtptri</i> COMPLEX for <i>ctptri</i> DOUBLE COMPLEX for <i>ztptri</i> . Array, size at least $\max(1, n(n+1)/2)$. Contains the packed triangular matrix A .

Output Parameters

<i>ap</i>	Overwritten by the packed n -by- n matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value. If <i>info</i> = i , the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *tptri* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\| \leq c(n)\varepsilon \|X\| \|A\|$$

$$\|X - A^{-1}\| \leq c(n)\varepsilon \|A^{-1}\| \|A\| \|X\|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?gseequ

Computes row and column scaling factors intended to equilibrate a general matrix and reduce its condition number.

Syntax

```
call sseequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dseequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cseequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zseequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call gseequ( a, r, c [,rowcnd] [,colcnd] [,amax] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate an m -by- n matrix *A* and reduce its condition number. The output array *r* returns the row scale factors and the array *c* the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix *B* with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See [?laqge](#) auxiliary function that uses scaling factors computed by [?gseequ](#).

Input Parameters

m INTEGER. The number of rows of the matrix *A*; $m \geq 0$.

n INTEGER. The number of columns of the matrix *A*; $n \geq 0$.

a REAL for *sgeequ*
 DOUBLE PRECISION for *dgeequ*
 COMPLEX for *cgeequ*
 DOUBLE COMPLEX for *zgeequ*.
 Array: size *lda* by *.
 Contains the *m*-by-*n* matrix *A* whose equilibration factors are to be computed.
 The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

Output Parameters

r, c REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays: *r* (size *m*), *c* (size *n*).
 If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*.
 If *info* = 0, the array *c* contains the column scale factors of the matrix *A*.

rowcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If *info* = 0, *colcnd* contains the ratio of the smallest *c*(*i*) to the largest *c*(*i*).

amax REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest element of the matrix *A*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, *i* > 0, and
i ≤ *m*, the *i*-th row of *A* is exactly zero;
i > *m*, the (*i*-*m*)th column of *A* is exactly zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gseequ` interface are as follows:

a	Holds the matrix A of size (m, n) .
r	Holds the vector of length (m) .
c	Holds the vector of length n .

Application Notes

All the components of r and c are restricted to be between `SMLNUM` = smallest safe number and `BIGNUM`= largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

`SMLNUM` and `BIGNUM` are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of `SMLNUM` and `BIGNUM` as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If $rowcnd \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by r .

If $colcnd \geq 0.1$, it is not worth scaling by c .

If $amax$ is very close to `SMLNUM` or very close to `BIGNUM`, the matrix A should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?gseequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a general matrix and reduce its condition number.

Syntax

```
call sgeeub( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

```
call dgeeub( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

```
call cgeeub( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

```
call zgeeub( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes row and column scalings intended to equilibrate an m -by- n general matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{i,j} = r(i) * a_{i,j} * c(j)$ have an absolute value of at most the radix.

$r(i)$ and $c(j)$ are restricted to be a power of the radix between $SMLNUM =$ smallest safe number and $BIGNUM =$ largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from `?geequ` by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

m INTEGER. The number of rows of the matrix A ; $m \geq 0$.

n INTEGER. The number of columns of the matrix A ; $n \geq 0$.

a REAL for `sgeequb`
 DOUBLE PRECISION for `dgeequb`
 COMPLEX for `cgeequb`
 DOUBLE COMPLEX for `zgeequb`.
 Array: size $(lda, *)$.
 Contains the m -by- n matrix A whose equilibration factors are to be computed.
 The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

r, c REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays: $r(m), c(n)$.
 If $info = 0$, or $info > m$, the array r contains the row scale factors for the matrix A .
 If $info = 0$, the array c contains the column scale factors for the matrix A .

rowcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If $info = 0$ or $info > m$, *rowcnd* contains the ratio of the smallest $r(i)$ to the largest $r(i)$. If $rowcnd \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by r .

colcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

If $info = 0$, $colcnd$ contains the ratio of the smallest $c(i)$ to the largest $c(i)$. If $colcnd \geq 0.1$, it is not worth scaling by c .

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix A . If $amax$ is very close to `SMLNUM` or very close to `BIGNUM`, the matrix should be scaled.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $i > 0$, and

$i \leq m$, the i -th row of A is exactly zero;

$i > m$, the $(i-m)$ -th column of A is exactly zero.

See Also

Error Analysis

Matrix Storage Schemes

?gbequ

Computes row and column scaling factors intended to equilibrate a banded matrix and reduce its condition number.

Syntax

```
call sgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

```
call dgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

```
call cgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

```
call zgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

```
call gbequ( ab, r, c [,kl] [,rowcnd] [,colcnd] [,amax] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes row and column scalings intended to equilibrate an m -by- n band matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See [?laqgb](#) auxiliary function that uses scaling factors computed by `?gbequ`.

Input Parameters

m INTEGER. The number of rows of the matrix A ; $m \geq 0$.

n INTEGER. The number of columns of the matrix A ; $n \geq 0$.

kl INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab REAL for *sgbequ*
 DOUBLE PRECISION for *dgbequ*
 COMPLEX for *cgbequ*
 DOUBLE COMPLEX for *zgbequ*.
 Array, size *ldab* by *. Contains the original band matrix *A* stored in rows from 1 to $kl + ku + 1$. The second dimension of *ab* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of *ab*; $ldab \geq kl + ku + 1$.

Output Parameters

r, c REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays: *r* (size *m*), *c* (size *n*).
 If *info* = 0, or *info* > *m*, the array *r* contains the row scale factors of the matrix *A*.
 If *info* = 0, the array *c* contains the column scale factors of the matrix *A*.

rowcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If *info* = 0 or *info* > *m*, *rowcnd* contains the ratio of the smallest *r*(*i*) to the largest *r*(*i*).

colcnd REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If *info* = 0, *colcnd* contains the ratio of the smallest *c*(*i*) to the largest *c*(*i*).

amax REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest element of the matrix *A*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i* and
 $i \leq m$, the *i*-th row of *A* is exactly zero;
 $i > m$, the (*i*-*m*)th column of *A* is exactly zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbequ` interface are as follows:

<code>ab</code>	Holds the array A of size $(kl+ku+1, n)$.
<code>r</code>	Holds the vector of length (m) .
<code>c</code>	Holds the vector of length n .
<code>kl</code>	If omitted, assumed $kl = ku$.
<code>ku</code>	Restored as $ku = lda-kl-1$.

Application Notes

All the components of r and c are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If $rowcnd \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by r .

If $colcnd \geq 0.1$, it is not worth scaling by c .

If $amax$ is very close to $SMLNUM$ or very close to $BIGNUM$, the matrix A should be scaled.

See Also

[Error Analysis](#)
[Matrix Storage Schemes](#)

?gbequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a banded matrix and reduce its condition number.

Syntax

```
call sgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes row and column scalings intended to equilibrate an m -by- n banded matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b(i,j)=r(i)*a(i,j)*c(j)$ have an absolute value of at most the radix.

$r(i)$ and $c(j)$ are restricted to be a power of the radix between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from `?gbequ` by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

m	INTEGER. The number of rows of the matrix A ; $m \geq 0$.
n	INTEGER. The number of columns of the matrix A ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
ab	REAL for <code>sgbequ</code> DOUBLE PRECISION for <code>dgbequ</code> COMPLEX for <code>cgbequ</code> DOUBLE COMPLEX for <code>zgbequ</code> . Array: size $ldab$ by * Contains the original banded matrix A stored in rows from 1 to $kl + ku + 1$. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(ku+1+i-j, j) = a(i, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$. The second dimension of ab must be at least $\max(1, n)$.
$ldab$	INTEGER. The leading dimension of a ; $ldab \geq \max(1, m)$.

Output Parameters

r, c	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: r (size m), c (size n). If $info = 0$, or $info > m$, the array r contains the row scale factors for the matrix A . If $info = 0$, the array c contains the column scale factors for the matrix A .
--------	---

<i>rowcnd</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>If $info = 0$ or $info > m$, <i>rowcnd</i> contains the ratio of the smallest $r(i)$ to the largest $r(i)$. If $rowcnd \geq 0.1$, and <i>amax</i> is neither too large nor too small, it is not worth scaling by r.</p>
<i>colcnd</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>If $info = 0$, <i>colcnd</i> contains the ratio of the smallest $c(i)$ to the largest $c(i)$. If $colcnd \geq 0.1$, it is not worth scaling by c.</p>
<i>amax</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Absolute value of the largest element of the matrix A. If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.</p>
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, the i-th diagonal element of A is nonpositive.</p> <p>$i \leq m$, the i-th row of A is exactly zero;</p> <p>$i > m$, the $(i-m)$-th column of A is exactly zero.</p>

See Also

Error Analysis
 Matrix Storage Schemes

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
call spoequ( n, a, lda, s, scnd, amax, info )
call dpoequ( n, a, lda, s, scnd, amax, info )
call cpoequ( n, a, lda, s, scnd, amax, info )
call zpoequ( n, a, lda, s, scnd, amax, info )
call poequ( a, s [,scnd] [,amax] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors such that $s(i) s[i + 1]$ contains

$$\frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $B_{i,j}=s(i) * A_{i,j} * s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsy](#) auxiliary function that uses scaling factors computed by [?poequ](#).

Input Parameters

n INTEGER. The order of the matrix A ; $n \geq 0$.

a REAL for `spoequ`
 DOUBLE PRECISION for `dpoequ`
 COMPLEX for `cpoequ`
 DOUBLE COMPLEX for `zpoequ`.
 Array: size *lda* by *.
 Contains the n -by- n symmetric or Hermitian positive definite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
 The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

s REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Array, size n .
 If *info* = 0, the array s contains the scale factors for A .

scond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If *info* = 0, *scond* contains the ratio of the smallest $s(i)$ to the largest $s(i)$.

amax REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest element of the matrix A .

info INTEGER.
 If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `poequ` interface are as follows:

a	Holds the matrix A of size (n, n) .
s	Holds the vector of length n .

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to `SMLNUM` or very close to `BIGNUM`, the matrix A should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?poequb

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
call spoequb( n, a, lda, s, scnd, amax, info )
call dpoequb( n, a, lda, s, scnd, amax, info )
call cpoequb( n, a, lda, s, scnd, amax, info )
call zpoequb( n, a, lda, s, scnd, amax, info )
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm).

These factors are chosen so that the scaled matrix B with elements $B_{i,j} = s(i) * A_{i,j} * s(j)$ has diagonal elements equal to 1. $s(i)$ is a power of two nearest to, but not exceeding $1/\sqrt{A_{i,i}}$.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

n	INTEGER. The order of the matrix A ; $n \geq 0$.
a	REAL for <code>spoequb</code>

DOUBLE PRECISION for dpoequb

COMPLEX for cpoequb

DOUBLE COMPLEX for zpoequb.

Array: size *lda* by *.

Contains the *n*-by-*n* symmetric or Hermitian positive definite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced.

The second dimension of *a* must be at least $\max(1, n)$.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *s*(*i*) to the largest *s*(*i*). If $scond \geq 0.1$, and *amax* is neither too large nor too small, it is not worth scaling by *s*.

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

call sppequ(*uplo*, *n*, *ap*, *s*, *scond*, *amax*, *info*)

call dppequ(*uplo*, *n*, *ap*, *s*, *scond*, *amax*, *info*)

```
call cppequ( uplo, n, ap, s, scond, amax, info )
call zppequ( uplo, n, ap, s, scond, amax, info )
call ppequ( ap, s [,scond] [,amax] [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). The output array s returns scale factors such that $s(i) s[i + 1]$ contains

, ! ! ! ! ! ! ! !

These factors are chosen so that the scaled matrix B with elements $b_{ij} = s(i) * a_{ij} * s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsp](#) auxiliary function that uses scaling factors computed by `?ppequ`.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> : If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>ap</i>	REAL for <code>sppequ</code> DOUBLE PRECISION for <code>dppequ</code> COMPLEX for <code>cppequ</code> DOUBLE COMPLEX for <code>zppequ</code> . Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
----------	---

Array, size (n).

If $info = 0$, the array s contains the scale factors for A .

$scond$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If $info = 0$, $scond$ contains the ratio of the smallest $s(i)$ to the largest $s(i)$.

$amax$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix A .

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ppequ` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

s

Holds the vector of length n .

$uplo$

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to `SMLNUM` or very close to `BIGNUM`, the matrix A should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

```
call spbequ( uplo, n, kd, ab, ldab, s, scnd, amax, info )
```

```
call dpbequ( uplo, n, kd, ab, ldab, s, scnd, amax, info )
```

```
call cpbequ( uplo, n, kd, ab, ldab, s, scnd, amax, info )
```

```
call zpbequ( uplo, n, kd, ab, ldab, s, scnd, amax, info )
```

```
call pbequ( ab, s [,scnd] [,amax] [,uplo] [,info] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite band matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors such that $s(i) s[i + 1]$ contains

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij} = s(i) * a_{ij} * s(j)$ has diagonal elements equal to 1. This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsb](#) auxiliary function that uses scaling factors computed by [?pbequ](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored in the array ab : If <i>uplo</i> = 'U', the array ab stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array ab stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>ab</i>	REAL for <code>spbequ</code> DOUBLE PRECISION for <code>dpbequ</code> COMPLEX for <code>cpbequ</code> DOUBLE COMPLEX for <code>zpbequ</code> . Array, size <i>ldab</i> by *. The array ap contains either the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>band storage</i> (see Matrix Storage Schemes). The second dimension of ab must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of the array ab ; $ldab \geq kd + 1$.

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbequ` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size (<i>kd</i> +1, <i>n</i>).
<i>s</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to `SMLNUM` or very close to `BIGNUM`, the matrix *A* should be scaled.

See Also

[Error Analysis](#)
[Matrix Storage Schemes](#)

?syequb

Computes row and column scaling factors intended to equilibrate a symmetric indefinite matrix and reduce its condition number.

Syntax

```
call ssyequb( uplo, n, a, lda, s, scnd, amax, work, info )
```

```
call dsyequb( uplo, n, a, lda, s, scond, amax, work, info )
call csyequb( uplo, n, a, lda, s, scond, amax, work, info )
call zsyequb( uplo, n, a, lda, s, scond, amax, work, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a symmetric indefinite matrix A and reduce its condition number (with respect to the two-norm).

The array s contains the scale factors, $s(i) = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix B with elements $b(i,j) = s(i) * a(i,j) * s(j)$ has ones on the diagonal.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored:
If *uplo* = 'U', the array a stores the upper triangular part of the matrix A .
If *uplo* = 'L', the array a stores the lower triangular part of the matrix A .

n INTEGER. The order of the matrix A ; $n \geq 0$.

a, work REAL for ssyequb
DOUBLE PRECISION for dsyequb
COMPLEX for csyequb
DOUBLE COMPLEX for zsyequb.
Array a : lda by *.
Contains the n -by- n symmetric indefinite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced. The second dimension of a must be at least $\max(1, n)$.
 $work(*)$ is a workspace array. The dimension of $work$ is at least $\max(1, 3*n)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

s REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, size (n).
If *info* = 0, the array s contains the scale factors for A .

<i>scond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>If $info = 0$, <i>scond</i> contains the ratio of the smallest $s(i)$ to the largest $s(i)$. If $scond \geq 0.1$, and <i>amax</i> is neither too large nor too small, it is not worth scaling by s.</p>
<i>amax</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Absolute value of the largest element of the matrix A. If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.</p>
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, the i-th diagonal element of A is nonpositive.</p>

See Also

[Error Analysis](#)
[Matrix Storage Schemes](#)

?heequb

Computes row and column scaling factors intended to equilibrate a Hermitian indefinite matrix and reduce its condition number.

Syntax

```
call cheequb( uplo, n, a, lda, s, scnd, amax, work, info )
call zheequb( uplo, n, a, lda, s, scnd, amax, work, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes row and column scalings intended to equilibrate a Hermitian indefinite matrix A and reduce its condition number (with respect to the two-norm).

The array s contains the scale factors, $s(i) = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix B with elements $b(i,j) = s(i) * a(i,j) * s(j)$ has ones on the diagonal.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A.</p>
-------------	---

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix *A*.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

a, work

COMPLEX for cheequb

DOUBLE COMPLEX for zheequb.

Array *a*: size *lda* by *.

Contains the *n*-by-*n* symmetric indefinite matrix *A* whose scaling factors are to be computed. Only the diagonal elements of *A* are referenced.

The second dimension of *a* must be at least $\max(1, n)$.

work(*) is a workspace array. The dimension of *work* is at least $\max(1, 3*n)$.

lda

INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

Output Parameters

s

REAL for cheequb

DOUBLE PRECISION for zheequb.

Array, size (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond

REAL for cheequb

DOUBLE PRECISION for zheequb.

If *info* = 0, *scond* contains the ratio of the smallest *s*(*i*) to the largest *s*(*i*). If $scond \geq 0.1$, and *amax* is neither too large nor too small, it is not worth scaling by *s*.

amax

REAL for cheequb

DOUBLE PRECISION for zheequb.

Absolute value of the largest element of the matrix *A*. If *amax* is very close to SMLNUM or BIGNUM, the matrix should be scaled.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

Error Analysis

Matrix Storage Schemes

Driver Routines

Table "Driver Routines for Solving Systems of Linear Equations" lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver	Expert Driver using Extra-Precise Iterative Refinement
general	?gesv	?gesvx	?gesvxx
general band	?gbsv	?gbsvx	?gbsvxx
general tridiagonal	?gtsv	?gtsvx	
diagonally dominant tridiagonal	?dtsvb		
symmetric/Hermitian positive-definite	?posv	?posvx	?posvxx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx	
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx	
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx	
symmetric/Hermitian indefinite	?sysv/?hesv ?sysv_rook/?hesv_rook	?sysvx/?hesvx	?sysvxx/?hesvxx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx	
complex symmetric	?sysv ?sysv_rook	?sysvx	
complex symmetric, packed storage	?spsv	?spsvx	

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of ?gesv and ?posv routines, the ? sign stands for combined character codes ds and zc for the mixed precision subroutines.

?gesv

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
```

```

call cgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call zgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dsgeev( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info )
call zcgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, rwork, iter, info )
call gesv( a, b [,ipiv] [,info] )

```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the system of linear equations $A*X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = P*L*U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $A*X = B$.

The `dsgeev` and `zcgesv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsgeev`) or single complex precision (`zcgesv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsgeev`) / double complex precision (`zcgesv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the right-hand sides:

```
rnmr < sqrt(n)*xnmr*anrm*eps*bwdmax
```

where

- `iter` is the number of the current iteration in the iterativerefinement process
- `rnmr` is the infinity-norm of the residual
- `xnmr` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<code>a</code>	REAL for <code>sgesv</code>

DOUBLE PRECISION for `dgesv` and `dsgesv`

COMPLEX for `cgsv`

DOUBLE COMPLEX for `zgesv` and `zcgesv`.

The array `a` (size `lda` by `*`) contains the n -by- n coefficient matrix A .

The second dimension of `a` must be at least $\max(1, n)$, the second dimension of `b` at least $\max(1, nrhs)$.

`b`

REAL for `sgsv`

DOUBLE PRECISION for `dgesv` and `dsgesv`

COMPLEX for `cgsv`

DOUBLE COMPLEX for `zgesv` and `zcgesv`.

The array `b` (size `ldb` by `*`) contains the n -by- $nrhs$ matrix of right hand side matrix B .

`lda`

INTEGER. The leading dimension of the array `a`; $lda \geq \max(1, n)$.

`ldb`

INTEGER. The leading dimension of the array `b`; $ldb \geq \max(1, n)$.

`ldx`

INTEGER. The leading dimension of the array `x`; $ldx \geq \max(1, n)$.

`work`

DOUBLE PRECISION for `dsgesv`

DOUBLE COMPLEX for `zcgesv`.

Workspace array, size at least $\max(1, n * nrhs)$. This array is used to hold the residual vectors.

`swork`

REAL for `dsgesv`

COMPLEX for `zcgesv`.

Workspace array, size at least $\max(1, n * (n + nrhs))$. This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.

`rwork`

DOUBLE PRECISION. Workspace array, size at least $\max(1, n)$.

Output Parameters

`a`

Overwritten by the factors L and U from the factorization of $A = P * L * U$; the unit diagonal elements of L are not stored.

If iterative refinement has been successfully used ($info = 0$ and $iter \geq 0$), then A is unchanged.

If double precision factorization has been used ($info = 0$ and $iter < 0$), then the array A contains the factors L and U from the factorization $A = P * L * U$; the unit diagonal elements of L are not stored.

`b`

Overwritten by the solution matrix X for `dgesv`, `sgsv`, `zgesv`, `zcgesv`. Unchanged for `dsgesv` and `zcgesv`.

<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The pivot indices that define the permutation matrix P ; row i of the matrix was interchanged with row $ipiv(i)$. Corresponds to the single precision factorization (if $info=0$ and $iter \geq 0$) or the double precision factorization (if $info=0$ and $iter < 0$).
<i>x</i>	DOUBLE PRECISION for <i>dsgesv</i> DOUBLE COMPLEX for <i>zcgsv</i> . Array, size <i>ldx</i> by <i>nrhs</i> . If $info = 0$, contains the n -by- $nrhs$ solution matrix X .
<i>iter</i>	INTEGER. If $iter < 0$: iterative refinement has failed, double precision factorization has been performed <ul style="list-style-type: none"> • If $iter = -1$: the routine fell back to full precision for implementation- or machine-specific reason • If $iter = -2$: narrowing the precision induced an overflow, the routine fell back to full precision • If $iter = -3$: failure of <i>sgetrf</i> for <i>dsgesv</i>, or <i>cgetrf</i> for <i>zcgsv</i> • If $iter = -31$: stop the iterative refinement after the 30th iteration. If $iter > 0$: iterative refinement has been successfully used. Returns the number of iterations.
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, $U_{i, i}$ (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *gesv* interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .

NOTE

Fortran 95 Interface is so far not available for the mixed precision subroutines *dsgesv/zcgsv*.

See Also

[ilaenv](#)

[?lamch](#)

[?getrf](#)

[Matrix Storage Schemes](#)

?gesvx

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )
```

```
call dgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )
```

```
call cgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )
```

```
call zgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )
```

```
call gesvx( a, b, x [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr] [,berr]
            [,rcond] [,rpvgrw] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A*X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gesvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors r and c are computed to equilibrate the system:

$$trans = 'N': \text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$$

$$trans = 'T': (\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

$$trans = 'C': (\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if `trans='N'`) or $\text{diag}(c) * B$ (if `trans = 'T'` or `'C'`).

2. If `fact = 'N'` or `'E'`, the LU decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as $A = P * L * U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n + 1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $\text{fact} = 'F'$: on entry, af and $ipiv$ contain the factored form of A. If equed is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. a, af, and $ipiv$ are not modified.</p> <p>If $\text{fact} = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $\text{fact} = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.</p>
<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $\text{trans} = 'N'$, the system has the form $A^*X = B$ (No transpose).</p> <p>If $\text{trans} = 'T'$, the system has the form $A^T X = B$ (Transpose).</p> <p>If $\text{trans} = 'C'$, the system has the form $A^H X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).</p>
<i>n</i>	<p>INTEGER. The number of linear equations; the order of the matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right hand sides; the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<i>a</i>	<p>REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx.</p> <p>The array a (size lda by $*$) contains the matrix A. If $\text{fact} = 'F'$ and equed is not 'N', then A must have been equilibrated by the scaling factors in r and/or c. The second dimension of a must be at least $\max(1, n)$.</p>
<i>af</i>	<p>REAL for sgesvx DOUBLE PRECISION for dgesvx COMPLEX for cgesvx DOUBLE COMPLEX for zgesvx.</p>

The array *afaf*(size *ldaf* by *) is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization $A = P * L * U$ as computed by ?getrf. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least $\max(1, n)$.

b
 REAL for sgesvx
 DOUBLE PRECISION for dgesvx
 COMPLEX for cgesvx
 DOUBLE COMPLEX for zgesvx.

The array *bb*(size *ldb* by *) contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

, *work*
 REAL for sgesvx
 DOUBLE PRECISION for dgesvx
 COMPLEX for cgesvx
 DOUBLE COMPLEX for zgesvx.

work(*) is a workspace array. The dimension of *work* must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.
 Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by ?getrf; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*) * *A* * *diag*(*c*).

r, c REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: r (size n), c (size n). The array r contains the row scale factors for A , and the array c contains the column scale factors for A . These arrays are input arguments if $fact = 'F'$ only; otherwise they are output arguments.

If $equed = 'R'$ or $'B'$, A is multiplied on the left by $diag(r)$; if $equed = 'N'$ or $'C'$, r is not accessed.

If $fact = 'F'$ and $equed = 'R'$ or $'B'$, each element of r must be positive.

If $equed = 'C'$ or $'B'$, A is multiplied on the right by $diag(c)$; if $equed = 'N'$ or $'R'$, c is not accessed.

If $fact = 'F'$ and $equed = 'C'$ or $'B'$, each element of c must be positive.

ldx INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

$iwork$ INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

$rwork$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, size at least $\max(1, 2*n)$; used in complex flavors only.

Output Parameters

x REAL for sgesvx
DOUBLE PRECISION for dgesvx
COMPLEX for cgesvx
DOUBLE COMPLEX for zgesvx.

Array, size ldx by $*$.

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the *equilibrated* system is:

$diag(C)^{-1} * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$;
 $diag(R)^{-1} * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$. The second dimension of x must be at least $\max(1, nrhs)$.

a Array a is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$. If $equed \neq 'N'$, A is scaled on exit as follows:

$equed = 'R'$: $A = diag(R) * A$

$equed = 'C'$: $A = A * diag(c)$

$equed = 'B'$: $A = diag(R) * A * diag(c)$.

<i>af</i>	If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the factors <i>L</i> and <i>U</i> from the factorization $A = PLU$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>a</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $diag(r)*B$ if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B'; overwritten by $diag(c)*B$ if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'C' or 'B'; not changed if <i>equed</i> = 'N'.
<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠ 'F'. See the description of <i>r, c</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>work, rwork</i>	On exit, <i>work</i> (1) for real flavors, or <i>rwork</i> (1) for complex flavors (the Fortran interface) contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If

$work(1)$ for real flavors, or $rwork(1)$ for complex flavors is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator $rcond$, and forward error bound $ferr$ could be unreliable. If factorization fails with $0 < info \leq n$, then $work(1)$ for real flavors, or $rwork(1)$ for complex flavors contains the reciprocal pivot growth factor for the leading $info$ columns of A .

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gesvx` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>r</i>	Holds the vector of length n . Default value for each element is $r(i) = 1.0_WP$.
<i>c</i>	Holds the vector of length n . Default value for each element is $c(i) = 1.0_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If $fact = 'F'$, then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.

rpvgrw

Real value that contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$.

See Also

Matrix Storage Schemes

?gesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides

Syntax

```
call sgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
            work, iwork, info )
```

```
call dgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
            work, iwork, info )
```

```
call cgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
            work, rwork, info )
```

```
call zgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
            work, rwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?gesvxx performs the following steps:

1. If *fact* = 'E', scaling factors r and c are computed to equilibrate the system:

$$\text{trans} = \text{'N'}: \text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$$

$$\text{trans} = \text{'T'}: (\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

$$\text{trans} = \text{'C'}: (\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans='N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).

2. If $fact = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = P*L*U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless `params(la_linrx_itref_i)` is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $fact = 'F'$, on entry, af and $ipiv$ contain the factored form of A. If $equed$ is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. Parameters a, af, and $ipiv$ are not modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.</p>
<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $trans = 'N'$, the system has the form $A*X = B$ (No transpose).</p> <p>If $trans = 'T'$, the system has the form $A^T*X = B$ (Transpose).</p> <p>If $trans = 'C'$, the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).</p>
<i>n</i>	<p>INTEGER. The number of linear equations; the order of the matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right hand sides; the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<i>a, af, b, work</i>	<p>REAL for <code>sgesvxx</code></p> <p>DOUBLE PRECISION for <code>dgesvxx</code></p> <p>COMPLEX for <code>cgesvxx</code></p> <p>DOUBLE COMPLEX for <code>zgesvxx</code>.</p> <p>Arrays: $a(\text{size } lda \text{ by } *)$, $af(\text{size } ldafby \text{ by } *)$, $b(\text{size } ldb \text{ by } *)$, $work(*)$.</p>

The array *a* contains the matrix *A*. If *fact* = 'F' and *equed* is not 'N', then *A* must have been equilibrated by the scaling factors in *r* and/or *c*. The second dimension of *a* must be at least $\max(1, n)$.

The array *af* is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization $A = P * L * U$ as computed by `?getrf`. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*. The second dimension of *af* must be at least $\max(1, n)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array. The dimension of *work* must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by `?getrf`; row *i* of the matrix was interchanged with row *ipiv*(*i*).

equed CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*) * *A* * *diag*(*c*).

r, c REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If $fact = 'F'$ and $equed = 'C'$ or $'B'$, each element of c must be positive.

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision. (Other values are reserved for future use.) <i>params</i> (2) : Maximum number of residual computations allowed for refinement. Default 10.0 Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for sgesvxx
 DOUBLE PRECISION for dgesvxx
 COMPLEX for cgesvxx
 DOUBLE COMPLEX for zgesvxx.

Array, size $(ldx, *)$.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:

$\text{inv}(\text{diag}(c)) * X$, if *trans* = 'N' and *equed* = 'C' or 'B'; or
 $\text{inv}(\text{diag}(r)) * X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least $\max(1, nrhs)$.

a Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': $A = \text{diag}(r) * A$
equed = 'C': $A = A * \text{diag}(c)$
equed = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$.

af If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization $A = PLU$ of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b Overwritten by $\text{diag}(r) * B$ if *trans* = 'N' and *equed* = 'R' or 'B';
 overwritten by *trans* = 'T' or 'C' and *equed* = 'C' or 'B';
 not changed if *equed* = 'N'.

r, c These arrays are output arguments if *fact* ≠ 'F'. Each element of these arrays is a power of the radix. See the description of *r, c* in *Input Arguments* section.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading $info$ columns of A . In `?gesvx`, this quantity is returned in `work(1)`.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size $nrhs$ by n_err_bnds . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first `(:,n_err_bnds)` entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are: $\ Z\ _{\infty} \cdot \ Z^{-1}\ _{\infty}$ <p>Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.</p>
<code>ipiv</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>ipiv</code> is an output argument and on exit contains the pivot indices from the factorization $A = P * L * U$ of the original matrix A (if <code>fact = 'N'</code>) or of the equilibrated matrix A (if <code>fact = 'E'</code>).
<code>equed</code>	If <code>fact ≠ 'F'</code> , then <code>equed</code> is an output argument. It specifies the form of equilibration that was done (see the description of <code>equed</code> in <i>Input Arguments</i> section).
<code>params</code>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. The solution to every right-hand side is guaranteed. If <code>info = -i</code> , the i -th parameter had an illegal value. If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; <code>rcond = 0</code> is returned. If <code>info = n+j</code> : The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is

reported. If a small componentwise error is not requested $params(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $err_bnds_norm(j,1) = 0.0$ or $err_bnds_comp(j,1) = 0.0$. See the definition of err_bnds_norm and err_bnds_comp for $err = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?gbsv

Computes the solution to the system of linear equations with a band coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call gbsv( ab, b [,kl] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n band matrix with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = L*U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals. The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

n	INTEGER. The order of A . The number of rows in B ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides. The number of columns in B ; $nrhs \geq 0$.
ab, b	REAL for sgbsv DOUBLE PRECISION for dgbsv COMPLEX for cgbsv DOUBLE COMPLEX for zgbsv.

Arrays: ab (size $ldab$ by $*$), b (size ldb by $*$).

The array ab contains the matrix A in band storage (see [Matrix Storage Schemes](#)). The second dimension of ab must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$ldab$ INTEGER. The leading dimension of the array ab . ($ldab \geq 2kl + ku + 1$)

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ab Overwritten by L and U . The diagonal and $kl + ku$ superdiagonals of U are stored in the first $1 + kl + ku$ rows of ab . The multipliers used to form L are stored in the next kl rows.

b Overwritten by the solution matrix X .

$ipiv$ INTEGER.

Array, size at least $\max(1, n)$. The pivot indices: row i was interchanged with row $ipiv(i)$.

$info$ INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $U(j, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbsv` interface are as follows:

ab Holds the array A of size $(2*kl+ku+1, n)$.

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector of length n .

kl If omitted, assumed $kl = ku$.

ku Restored as $ku = lda - 2*kl - 1$.

See Also

[Matrix Storage Schemes](#)

?gbsvx

Computes the solution to the real or complex system of linear equations with a band coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call dgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call cgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call gbsvx( ab, b, x [,kl] [,afb] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr]
[,berr] [,rcond] [,rpvgrw] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A*X = B$, $A^T*X = B$, or $A^H*X = B$, where A is a band matrix of order n with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gbsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors r and c are computed to equilibrate the system:

$$trans = 'N': diag(r)*A*diag(c) *inv(diag(c))*X = diag(r)*B$$

$$trans = 'T': (diag(r)*A*diag(c))^T *inv(diag(r))*X = diag(c)*B$$

$$trans = 'C': (diag(r)*A*diag(c))^H *inv(diag(r))*X = diag(c)*B$$

Whether the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans = 'N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).

2. If $fact = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = L*U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F': on entry, <i>afb</i> and <i>ipiv</i> contain the factored form of A. If <i>equed</i> is not 'N', the matrix A is equilibrated with scaling factors given by r and c. <i>ab</i>, <i>afb</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix A will be copied to <i>afb</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix A will be equilibrated if necessary, then copied to <i>afb</i> and factored.</p>
<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A^*X = B$ (No transpose).</p> <p>If <i>trans</i> = 'T', the system has the form $A^T X = B$ (Transpose).</p> <p>If <i>trans</i> = 'C', the system has the form $A^H X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).</p>
<i>n</i>	<p>INTEGER. The number of linear equations, the order of the matrix A; $n \geq 0$.</p>
<i>kl</i>	<p>INTEGER. The number of subdiagonals within the band of A; $kl \geq 0$.</p>
<i>ku</i>	<p>INTEGER. The number of superdiagonals within the band of A; $ku \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right hand sides, the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<i>ab</i> , <i>afb</i> , <i>b</i> , <i>work</i>	<p>REAL for <i>sgbsvx</i></p> <p>DOUBLE PRECISION for <i>dgbsvx</i></p> <p>COMPLEX for <i>cgbsvx</i></p> <p>DOUBLE COMPLEX for <i>zgbsvx</i>.</p> <p>Arrays: <i>ab</i>(<i>ldab</i>, *), <i>afb</i>(<i>ldaafb</i>, *), <i>b</i>(size <i>ldb</i> by *), <i>work</i>(*).</p> <p>The array <i>ab</i> contains the matrix A in band storage (see Matrix Storage Schemes). The second dimension of <i>ab</i> must be at least $\max(1, n)$. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then A must have been equilibrated by the scaling factors in r and/or c.</p> <p>The array <i>afb</i> is an input argument if <i>fact</i> = 'F'. The second dimension of <i>afb</i> must be at least $\max(1, n)$. It contains the factored form of the matrix A, that is, the factors L and U from the factorization $A = P^*L^*U$ as computed by ?gbtrf. U is stored as an upper triangular band matrix with $kl + ku$ superdiagonals in the first $1 + kl + ku$ rows of <i>afb</i>. The multipliers used during the factorization are stored in the next kl rows. If <i>equed</i> is not 'N', then <i>afb</i> is the factored form of the equilibrated matrix A.</p>

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work()* is a workspace array. The dimension of *work* must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

ldab INTEGER. The leading dimension of *ab*; $ldab \geq kl + ku + 1$.

ldafb INTEGER. The leading dimension of *afb*; $ldafb \geq 2*kl + ku + 1$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization $A = P*L*U$ as computed by *?gbtrf*; row *i* of the matrix was interchanged with row *ipiv(i)*.

equed CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag(r)*.

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag(c)*.

if *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag(r) * A * diag(c)*.

r, c REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r* (size *n*), *c* (size *n*).

The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag(r)*; if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag(c)*; if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Workspace array, size at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for <i>sgbsvx</i> DOUBLE PRECISION for <i>dgbvsx</i> COMPLEX for <i>cgbvsx</i> DOUBLE COMPLEX for <i>zgbvsx</i> . Array, size <i>ldx</i> by *. If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠ 'N', and the solution to the <i>equilibrated</i> system is: $\text{inv}(\text{diag}(c)) * X$, if <i>trans</i> = 'N' and <i>equed</i> = 'C' or 'B'; $\text{inv}(\text{diag}(r)) * X$, if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'R' or 'B'. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
<i>ab</i>	Array <i>ab</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>equed</i> ≠ 'N', <i>A</i> is scaled on exit as follows: <i>equed</i> = 'R': $A = \text{diag}(r) * A$ <i>equed</i> = 'C': $A = A * \text{diag}(c)$ <i>equed</i> = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$.
<i>afb</i>	If <i>fact</i> = 'N' or 'E', then <i>afb</i> is an output argument and on exit returns details of the <i>LU</i> factorization of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ab</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(r) * b$ if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B'; overwritten by $\text{diag}(c) * b$ if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'C' or 'B'; not changed if <i>equed</i> = 'N'.
<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠ 'F'. See the description of <i>r, c</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done).

If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

ipiv

If $fact = 'N'$ or $'E'$, then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = L*U$ of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$).

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n+1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, rwork

On exit, *work*(1) for real flavors, or *rwork*(1) for complex flavors, contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If *work*(1) for real flavors, or *rwork*(1) for complex flavors is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator $rcond$, and forward error bound *ferr* could be unreliable. If factorization fails with

$0 < info \leq n$, then $work(1)$ for real flavors, or $rwork(1)$ for complex flavors contains the reciprocal pivot growth factor for the leading $info$ columns of A .

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n+1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbsvx` interface are as follows:

<i>ab</i>	Holds the array A of size $(kl+ku+1, n)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>afb</i>	Holds the array AF of size $(2*kl+ku+1, n)$.
<i>ipiv</i>	Holds the vector of length n .
<i>r</i>	Holds the vector of length n . Default value for each element is $r(i) = 1.0_WP$.
<i>c</i>	Holds the vector of length n . Default value for each element is $c(i) = 1.0_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If $fact = 'F'$, then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>rpvgrw</i>	Real value that contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$.

<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda - kl - 1$.

See Also

Matrix Storage Schemes

?gbsvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a banded coefficient matrix A and multiple right-hand sides

Syntax

```
call sgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, iwork, info )
```

```
call dgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, iwork, info )
```

```
call cgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, rwork, info )
```

```
call zgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, rwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, $A^T \cdot X = B$, or $A^H \cdot X = B$, where A is an n -by- n banded matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?gbsvxx performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

$$\text{trans} = \text{'N'}: \text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$$

$$\text{trans} = \text{'T'}: (\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

$$\text{trans} = \text{'C'}: (\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans='N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).

2. If $fact = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = P*L*U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $params(1)$ is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $fact = 'F'$, on entry, <i>afb</i> and <i>ipiv</i> contain the factored form of A. If <i>equed</i> is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. Parameters <i>ab</i>, <i>afb</i>, and <i>ipiv</i> are not modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to <i>afb</i> and factored.</p> <p>If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to <i>afb</i> and factored.</p>
<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $trans = 'N'$, the system has the form $A*X = B$ (No transpose).</p> <p>If $trans = 'T'$, the system has the form $A^T*X = B$ (Transpose).</p> <p>If $trans = 'C'$, the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).</p>
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>ab, afb, b, work</i>	<p>REAL for <i>sgbsvxx</i></p> <p>DOUBLE PRECISION for <i>dgbsvxx</i></p> <p>COMPLEX for <i>cgbsvxx</i></p>

DOUBLE COMPLEX for zgbsvxx.

Arrays: $ab(ldab, *)$, $afb(ldafb, *)$, $b(\text{size } ldb \text{ by } *)$, $work(*)$.

The array ab contains the matrix A in band storage, in rows 1 to $kl+ku+1$. The j -th column of A is stored in the j -th column of the array ab as follows:

$$ab(ku+1+i-j, j) = A(i, j) \text{ for } \max(1, j-ku) \leq i \leq \min(n, j+kl).$$

If $fact = 'F'$ and $equed$ is not 'N', then AB must have been equilibrated by the scaling factors in r and/or c . The second dimension of a must be at least $\max(1, n)$.

The array afb is an input argument if $fact = 'F'$. It contains the factored form of the banded matrix A , that is, the factors L and U from the factorization $A = P*L*U$ as computed by `?gbtrf`. U is stored as an upper triangular banded matrix with $kl + ku$ superdiagonals in rows 1 to $kl + ku + 1$. The multipliers used during the factorization are stored in rows $kl + ku + 2$ to $2*kl + ku + 1$. If $equed$ is not 'N', then afb is the factored form of the equilibrated matrix A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

$ldab$ INTEGER. The leading dimension of the array ab ; $ldab \geq kl+ku+1$.

$ldafb$ INTEGER. The leading dimension of the array afb ; $ldafb \geq 2*kl+ku+1$.

$ipiv$ INTEGER.

Array, size at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains the pivot indices from the factorization $A = P*L*U$ as computed by `?gbtrf`; row i of the matrix was interchanged with row $ipiv(i)$.

$equed$ CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

If $equed = 'R'$, row equilibration was done, that is, A has been premultiplied by $diag(r)$.

If $equed = 'C'$, column equilibration was done, that is, A has been postmultiplied by $diag(c)$.

If $equed = 'B'$, both row and column equilibration was done, that is, A has been replaced by $diag(r)*A*diag(c)$.

r, c REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: r (size n), c (size n). The array r contains the row scale factors for A , and the array c contains the column scale factors for A . These arrays are input arguments if $fact = 'F'$ only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag(r)*; if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag(c)*; if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors). =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the extra-precise refinement algorithm. (Other values are reserved for future use.) <i>params</i> (2) : Maximum number of residual computations allowed for refinement. Default 10.0 Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a

technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 2*n)$; used in complex flavors only.

Output Parameters

x REAL for *sgbsvxx*
 DOUBLE PRECISION for *dgbsvxx*
 COMPLEX for *cgbsvxx*
 DOUBLE COMPLEX for *zgbsvxx*.
 Array, size *ldx* by *.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:

$\text{inv}(\text{diag}(c)) * X$, if *trans* = 'N' and *equed* = 'C' or 'B'; or
 $\text{inv}(\text{diag}(r)) * X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least $\max(1, nrhs)$.

ab Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': $A = \text{diag}(r) * A$
equed = 'C': $A = A * \text{diag}(c)$
equed = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$.

afb If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns the factors *L* and *U* from the factorization $A = PLU$ of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

b Overwritten by $\text{diag}(r) * B$ if *trans* = 'N' and *equed* = 'R' or 'B';
 overwritten by *trans* = 'T' or 'C' and *equed* = 'C' or 'B';
 not changed if *equed* = 'N'.

<i>r, c</i>	<p>These arrays are output arguments if <i>fact</i> ≠ 'F'. Each element of these arrays is a power of the radix. See the description of <i>r, c</i> in <i>Input Arguments</i> section.</p>
<i>rcond</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.</p>
<i>rpvgrw</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Contains the reciprocal pivot growth factor:</p> $\ A\ /\ U\ $ <p>If this is much less than 1, the stability of the <i>LU</i> factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>X</i>, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i>. In <i>?gbsvx</i>, this quantity is returned in <i>work(1)</i>.</p>
<i>berr</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>
<i>err_bnds_norm</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array of size <i>nrhs</i> by <i>n_err_bnds</i>. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:</p> <p>Normwise relative error in the <i>i</i>-th solution vector</p> $\frac{\max_j X_{true_{ji}} - X_{ji} }{\max_j X_{ji} }$ <p>The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.</p> <p>The first index in <i>err_bnds_norm(i, :)</i> corresponds to the <i>i</i>-th right-hand side.</p> <p>The second index in <i>err_bnds_norm(:, err)</i> contains the following three fields:</p>

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first `(:,n_err_bnds)` entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

<code>ipiv</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>ipiv</code> is an output argument and on exit contains the pivot indices from the factorization $A = P * L * U$ of the original matrix A (if <code>fact = 'N'</code>) or of the equilibrated matrix A (if <code>fact = 'E'</code>).
<code>equed</code>	If <code>fact ≠ 'F'</code> , then <code>equed</code> is an output argument. It specifies the form of equilibration that was done (see the description of <code>equed</code> in <i>Input Arguments</i> section).
<code>params</code>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. The solution to every right-hand side is guaranteed. If <code>info = -i</code> , the i -th parameter had an illegal value. If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; <code>rcond = 0</code> is returned.

If $info = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $params(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $err_bnds_norm(j,1) = 0.0$ or $err_bnds_comp(j,1) = 0.0$. See the definition of err_bnds_norm and err_bnds_comp for $err = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?gtsv

Computes the solution to the system of linear equations with a tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sgtsv( n, nrhs, dl, d, du, b, ldb, info )
call dgtsv( n, nrhs, dl, d, du, b, ldb, info )
call cgtsv( n, nrhs, dl, d, du, b, ldb, info )
call zgtsv( n, nrhs, dl, d, du, b, ldb, info )
call gtsv( dl, d, du, b [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the system of linear equations $A^*X = B$, where A is an n -by- n tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T * X = B$ may be solved by interchanging the order of the arguments du and dl .

Input Parameters

n	INTEGER. The order of A , the number of rows in B ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
dl	REAL for sgtsv DOUBLE PRECISION for dgtsv COMPLEX for cgtsv DOUBLE COMPLEX for zgtsv. The array dl (size $n - 1$) contains the $(n - 1)$ subdiagonal elements of A .

<i>d</i>	<p>REAL for <code>sgtsv</code></p> <p>DOUBLE PRECISION for <code>dgtsv</code></p> <p>COMPLEX for <code>cgtsv</code></p> <p>DOUBLE COMPLEX for <code>zgtsv</code>.</p> <p>The array <i>d</i> (size <i>n</i>) contains the diagonal elements of <i>A</i>.</p>
<i>du</i>	<p>REAL for <code>sgtsv</code></p> <p>DOUBLE PRECISION for <code>dgtsv</code></p> <p>COMPLEX for <code>cgtsv</code></p> <p>DOUBLE COMPLEX for <code>zgtsv</code>.</p> <p>The array <i>du</i> (size <i>n</i> - 1) contains the (<i>n</i> - 1) superdiagonal elements of <i>A</i>.</p>
<i>b</i>	<p>REAL for <code>sgtsv</code></p> <p>DOUBLE PRECISION for <code>dgtsv</code></p> <p>COMPLEX for <code>cgtsv</code></p> <p>DOUBLE COMPLEX for <code>zgtsv</code>.</p> <p>The array <i>b</i>(size <i>ldb</i> by *) contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least <code>max(1, nrhs)</code>.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; $ldb \geq \max(1, n)$.</p>

Output Parameters

<i>dl</i>	<p>Overwritten by the (<i>n</i>-2) elements of the second superdiagonal of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i>. These elements are stored in <code>dl(1), ..., dl(n - 2)</code>.</p>
<i>d</i>	<p>Overwritten by the <i>n</i> diagonal elements of <i>U</i>.</p>
<i>du</i>	<p>Overwritten by the (<i>n</i>-1) elements of the first superdiagonal of <i>U</i>.</p>
<i>b</i>	<p>Overwritten by the solution matrix <i>X</i>.</p>
<i>info</i>	<p>INTEGER. If <code>info = 0</code>, the execution is successful.</p> <p>If <code>info = -i</code>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <code>info = i</code>, <i>U</i>(<i>i</i>, <i>i</i>) is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gtsv` interface are as follows:

<i>dl</i>	<p>Holds the vector of length (<i>n</i>-1).</p>
-----------	---

<i>d</i>	Holds the vector of length <i>n</i> .
<i>dl</i>	Holds the vector of length (<i>n</i> -1).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).

See Also

Matrix Storage Schemes

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal coefficient matrix *A* and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call sgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )
```

```
call dgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )
```

```
call cgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )
```

```
call zgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )
```

```
call gtsvx( dl, d, du, b, x [,dlf] [,df] [,duf] [,du2] [,ipiv] [,fact] [,trans] [,ferr]
            [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, $A^T*X = B$, or $A^H*X = B$, where *A* is a tridiagonal matrix of order *n*, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:

1. If *fact* = 'N', the *LU* decomposition is used to factor the matrix *A* as $A = L*U$, where *L* is a product of permutation and unit lower bidiagonal matrices and *U* is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.
2. If some $U_{i,i} = 0$, so that *U* is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than machine precision, *info* = *n* + 1 is returned as a warning, but the routine still goes on to solve for *X* and compute error bounds as described below.
3. The system of equations is solved for *X* using the factored form of *A*.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F' or 'N'.</p> <p>Specifies whether or not the factored form of the matrix A has been supplied on entry.</p> <p>If $fact = 'F'$: on entry, dlf, df, duf, $du2$, and $ipiv$ contain the factored form of A; arrays dl, d, du, dlf, df, duf, $du2$, and $ipiv$ will not be modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to dlf, df, and duf and factored.</p>
<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $trans = 'N'$, the system has the form $A^*X = B$ (No transpose).</p> <p>If $trans = 'T'$, the system has the form $A^T*X = B$ (Transpose).</p> <p>If $trans = 'C'$, the system has the form $A^H*X = B$ (Conjugate transpose).</p>
<i>n</i>	<p>INTEGER. The number of linear equations, the order of the matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right hand sides, the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<i>dl,d,du,dlf,df, duf,du2,b,work</i>	<p>REAL for <code>sgtsvx</code></p> <p>DOUBLE PRECISION for <code>dgtsvx</code></p> <p>COMPLEX for <code>cgtsvx</code></p> <p>DOUBLE COMPLEX for <code>zgtsvx</code>.</p> <p>Arrays:</p> <p>dl, size $(n - 1)$, contains the subdiagonal elements of A.</p> <p>d, size (n), contains the diagonal elements of A.</p> <p>du, size $(n - 1)$, contains the superdiagonal elements of A.</p> <p>dlf, size $(n - 1)$. If $fact = 'F'$, then dlf is an input argument and on entry contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A as computed by <code>?gttrf</code>.</p> <p>df, size (n). If $fact = 'F'$, then df is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A.</p> <p>duf, size $(n - 1)$. If $fact = 'F'$, then duf is an input argument and on entry contains the $(n - 1)$ elements of the first superdiagonal of U.</p> <p>$du2$, size $(n - 2)$. If $fact = 'F'$, then $du2$ is an input argument and on entry contains the $(n - 2)$ elements of the second superdiagonal of U.</p> <p>$b(l_{db}, *)$ contains the right-hand side matrix B. The second dimension of b must be at least $\max(1, nrhs)$.</p>

work(*) is a workspace array. The size of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. If *fact* = 'F', then *ipiv* is an input argument and on entry contains the pivot indices, as returned by [?gttrf](#).

iwork INTEGER. Workspace array, size (*n*). Used for real flavors only.

rwork REAL for *cgtsvx*

DOUBLE PRECISION for *zgtsvx*.

Workspace array, size (*n*). Used for complex flavors only.

Output Parameters

x REAL for *sgtsvx*

DOUBLE PRECISION for *dgtsvx*

COMPLEX for *cgtsvx*

DOUBLE COMPLEX for *zgtsvx*.

Array, size *ldx* by *.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X*. The second dimension of *x* must be at least $\max(1, nrhs)$.

dlf If *fact* = 'N', then *dlf* is an output argument and on exit contains the (*n*-1) multipliers that define the matrix *L* from the *LU* factorization of *A*.

df If *fact* = 'N', then *df* is an output argument and on exit contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf If *fact* = 'N', then *duf* is an output argument and on exit contains the (*n*-1) elements of the first superdiagonal of *U*.

du2 If *fact* = 'N', then *du2* is an output argument and on exit contains the (*n*-2) elements of the second superdiagonal of *U*.

ipiv The array *ipiv* is an output argument if *fact* = 'N' and, on exit, contains the pivot indices from the factorization $A = L*U$; row *i* of the matrix was interchanged with row *ipiv*(*i*). The value of *ipiv*(*i*) will always be *i* or *i*+1; *ipiv*(*i*)=*i* indicates a row interchange was not required.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A . If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

*info*INTEGER. If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has not been completed unless $i = n$, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gtsvx` interface are as follows:

<i>d1</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.

<i>dlf</i>	Holds the vector of length $(n-1)$.
<i>df</i>	Holds the vector of length n .
<i>duf</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then the arguments <i>dlf</i> , <i>df</i> , <i>duf</i> , <i>du2</i> , and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

See Also

Matrix Storage Schemes

?dtsvb

Computes the solution to the system of linear equations with a diagonally dominant tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call ddtsvb( n, nrhs, dl, d, du, b, ldb, info )
call cdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call zdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call dtsvb( dl, d, du, b [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?dtsvb routine solves a system of linear equations $A^*X = B$ for X , where A is an n -by- n diagonally dominant tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses the BABE (Burning At Both Ends) algorithm.

Note that the equation $A^T*X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

<i>n</i>	INTEGER. The order of A , the number of rows in B ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

dl, d, du, b REAL for `sdtsvb`
 DOUBLE PRECISION for `ddtsvb`
 COMPLEX for `cdtsvb`
 DOUBLE COMPLEX for `zdtsvb`.

Arrays: *dl* (size $n - 1$), *d* (size n), *du* (size $n - 1$), *b*(size $ldb, *$).
 The array *dl* contains the $(n - 1)$ subdiagonal elements of *A*.
 The array *d* contains the diagonal elements of *A*.
 The array *du* contains the $(n - 1)$ superdiagonal elements of *A*.
 The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

Output Parameters

dl Overwritten by the $(n-1)$ elements of the subdiagonal of the lower triangular matrices L_1, L_2 from the factorization of *A* (see [dttrfb](#)).

d Overwritten by the n diagonal element reciprocals of *U*.

b Overwritten by the solution matrix *X*.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = $-i$, the i -th parameter had an illegal value.
 If *info* = i , u_{ii} is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i :
 $1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems have no numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gtsv](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
 - Applying the `?dtsvb` factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.
-

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A and multiple right-hand sides.

Syntax

```
call sposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dposv( uplo, n, nrhs, a, lda, b, ldb, info )
call cposv( uplo, n, nrhs, a, lda, b, ldb, info )
call zposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dsposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, iter, info )
call zcposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, rwork, iter, info )
call posv( a, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n symmetric/Hermitian positive-definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T*U$ (real flavors) and $A = U^H*U$ (complex flavors), if $uplo = 'U'$

or $A = L*L^T$ (real flavors) and $A = L*L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A*X = B$.

The `dsposv` and `zcposv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsposv`) or single complex precision (`zcposv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsposv`) / double complex precision (`zcposv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision/complex performance over double precision/double complex performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

`iter > itermax`

or for all the right-hand sides:

`rnrm < sqrt(n)*xnrm*anrm*eps*bwdmax,`

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnrm` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution

- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <code>uplo</code> = 'U', the upper triangle of A is stored. If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<code>a, b</code>	REAL for <code>sposv</code> DOUBLE PRECISION for <code>dposv</code> and <code>dsposv</code> . COMPLEX for <code>cposv</code> DOUBLE COMPLEX for <code>zposv</code> and <code>zcposv</code> . Arrays: <code>a(size lda,*)</code> , <code>b(ldb,*)</code> . The array <code>a</code> contains the upper or the lower triangular part of the matrix A (see <code>uplo</code>). The second dimension of <code>a</code> must be at least $\max(1, n)$. Note that in the case of <code>zcposv</code> the imaginary parts of the diagonal elements need not be set and are assumed to be zero. The array <code>b</code> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <code>b</code> must be at least $\max(1, nrhs)$.
<code>lda</code>	INTEGER. The leading dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of <code>b</code> ; $ldb \geq \max(1, n)$.
<code>ldx</code>	INTEGER. The leading dimension of the array <code>x</code> ; $ldx \geq \max(1, n)$.
<code>work</code>	DOUBLE PRECISION for <code>dsposv</code> DOUBLE COMPLEX for <code>zcposv</code> . Workspace array, size $(n * nrhs)$. This array is used to hold the residual vectors.
<code>swork</code>	REAL for <code>dsgesv</code> COMPLEX for <code>zcgesv</code> . Workspace array, size $(n * (n + nrhs))$. This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.
<code>rwork</code>	DOUBLE PRECISION. Workspace array, size (n) .

Output Parameters

<i>a</i>	<p>If <i>info</i> = 0, the upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i>, as specified by <i>uplo</i>.</p> <p>If iterative refinement has been successfully used (<i>info</i>= 0 and <i>iter</i>≥ 0), then <i>A</i> is unchanged.</p> <p>If double precision factorization has been used (<i>info</i>= 0 and <i>iter</i> < 0), then the array <i>A</i> contains the factors <i>L</i> or <i>U</i> from the Cholesky factorization.</p>
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. The pivot indices that define the permutation matrix <i>P</i>; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i>(<i>i</i>). Corresponds to the single precision factorization (if <i>info</i>= 0 and <i>iter</i>≥ 0) or the double precision factorization (if <i>info</i>= 0 and <i>iter</i> < 0).</p>
<i>x</i>	<p>DOUBLE PRECISION for dsposv</p> <p>DOUBLE COMPLEX for zcposv.</p> <p>Array, size <i>ldx</i> by <i>nrhs</i>. If <i>info</i> = 0, contains the <i>n</i>-by-<i>nrhs</i> solution matrix <i>X</i>.</p>
<i>iter</i>	<p>INTEGER.</p> <p>If <i>iter</i> < 0: iterative refinement has failed, double precision factorization has been performed</p> <ul style="list-style-type: none"> • If <i>iter</i> = -1: the routine fell back to full precision for implementation- or machine-specific reason • If <i>iter</i> = -2: narrowing the precision induced an overflow, the routine fell back to full precision • If <i>iter</i> = -3: failure of <i>spotrf</i> for dsposv, or <i>cpotrf</i> for zcposv • If <i>iter</i> = -31: stop the iterative refinement after the 30th iteration. <p>If <i>iter</i> > 0: iterative refinement has been successfully used. Returns the number of iterations.</p>
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `posv` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call sposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )
```

```
call dposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )
```

```
call cposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
```

```
call zposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
```

```
call posvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{inv}(\text{diag}(s))*X = \text{diag}(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s)*A*\text{diag}(s)$ and B by $\text{diag}(s)*B$.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U^T*U \text{ (real)}, A = U^H*U \text{ (complex)}, \text{ if } \text{uplo} = \text{'U'},$$

$$\text{or } A = L*L^T \text{ (real)}, A = L*L^H \text{ (complex)}, \text{ if } \text{uplo} = \text{'L'},$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F': on entry, <i>af</i> contains the factored form of A. If <i>equed</i> = 'Y', the matrix A has been equilibrated with scaling factors given by s. <i>a</i> and <i>af</i> will not be modified.</p> <p>If <i>fact</i> = 'N', the matrix A will be copied to <i>af</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix A will be equilibrated if necessary, then copied to <i>af</i> and factored.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of A is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of A is stored.</p>
<i>n</i>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.</p>
<i>a, af, b, work</i>	<p>REAL for <i>sposvx</i></p> <p>DOUBLE PRECISION for <i>dposvx</i></p> <p>COMPLEX for <i>cposvx</i></p> <p>DOUBLE COMPLEX for <i>zposvx</i>.</p> <p>Arrays: <i>a</i>(size <i>lda</i> by *), <i>af</i>(size <i>ldaf</i> by *), <i>b</i>(size <i>ldb</i> by *), <i>work</i>(*).</p> <p>The array <i>a</i> contains the matrix A as specified by <i>uplo</i>. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', then A must have been equilibrated by the scaling factors in s, and <i>a</i> must contain the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>

The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix $diag(s)*A*diag(s)$. The second dimension of *af* must be at least $\max(1, n)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array. The dimension of *work* must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

equed CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by $diag(s)*A*diag(s)$.

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for cposvx

DOUBLE PRECISION for zposvx.

Workspace array, size at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x REAL for sposvx

DOUBLE PRECISION for dposvx

COMPLEX for `cposvx`

DOUBLE COMPLEX for `zposvx`.

Array, size `ldx` by `*`.

If `info = 0` or `info = n+1`, the array `x` contains the solution matrix `X` to the *original* system of equations. Note that if `equed = 'Y'`, `A` and `B` are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$. The second dimension of `x` must be at least $\max(1, \text{nrhs})$.

`a`

Array `a` is not modified on exit if `fact = 'F'` or `'N'`, or if `fact = 'E'` and `equed = 'N'`.

If `fact = 'E'` and `equed = 'Y'`, `A` is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

`af`

If `fact = 'N'` or `'E'`, then `af` is an output argument and on exit returns the triangular factor `U` or `L` from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix `A` (if `fact = 'N'`), or of the equilibrated matrix `A` (if `fact = 'E'`). See the description of `a` for the form of the equilibrated matrix.

`b`

Overwritten by $\text{diag}(s) * B$, if `equed = 'Y'`; not changed if `equed = 'N'`.

`s`

This array is an output argument if `fact ≠ 'F'`. See the description of `s` in *Input Arguments* section.

`rcond`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix `A` after equilibration (if done). If `rcond` is less than the machine precision (in particular, if `rcond = 0`), the matrix is singular to working precision. This condition is indicated by a return code of `info > 0`.

`ferr`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector x_j (the j -th column of the solution matrix `X`). If `xtrue` is the true solution corresponding to x_j , `ferr(j)` is an estimated upper bound for the magnitude of the largest element in $(x_j) - xtrue$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for `rcond`, and is almost always a slight overestimate of the true error.

`berr`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \leq n$, the leading minor of order *i* (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and $i = n + 1$, then U is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `posvx` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>s</i>	Holds the vector of length n . Default value for each element is $s(i) = 1.0_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

See Also

[Matrix Storage Schemes](#)

?posvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A applying the Cholesky factorization.

Syntax

```
call sposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
```

```
call dposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
```

```
call cposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

```
call zposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is an n -by- n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?posvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s) * \text{inv}(\text{diag}(s))*X = \text{diag}(s)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s)*A*\text{diag}(s)$ and B by $\text{diag}(s)*B$.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U^T*U \text{ (real)}, A = U^H*U \text{ (complex)}, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L*L^T \text{ (real)}, A = L*L^H \text{ (complex)}, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $params(1)$ is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $fact = 'F'$, on entry, af contains the factored form of A. If $equed$ is not 'N', the matrix A has been equilibrated with scaling factors given by s. Parameters a and af are not modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If $uplo = 'U'$, the upper triangle of A is stored.</p> <p>If $uplo = 'L'$, the lower triangle of A is stored.</p>
<i>n</i>	<p>INTEGER. The number of linear equations; the order of the matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides; the number of columns of the matrices B and X; $nrhs \geq 0$.</p>
<i>a, af, b, work</i>	<p>REAL for <code>sposvxx</code></p> <p>DOUBLE PRECISION for <code>dposvxx</code></p> <p>COMPLEX for <code>cposvxx</code></p> <p>DOUBLE COMPLEX for <code>zposvxx</code>.</p> <p>Arrays: $a(\text{size } lda \text{ by } *)$, $af(\text{size } ldafby \text{ by } *)$, $b(\text{size } ldb \text{ by } *)$, $work(*)$.</p> <p>The array a contains the matrix A as specified by $uplo$. If $fact = 'F'$ and $equed = 'Y'$, then A must have been equilibrated by the scaling factors in s, and a must contain the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of a must be at least $\max(1, n)$.</p> <p>The array af is an input argument if $fact = 'F'$. It contains the triangular factor U or L from the Cholesky factorization of A in the same storage format as A. If $equed$ is not 'N', then af is the factored form of the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of af must be at least $\max(1, n)$.</p>

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array. The dimension of *work* must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of the array <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N'). if <i>equed</i> = 'Y', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag(s)*A*diag(s)</i> .
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i> . This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument. If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive. Each element of <i>s</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in the <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for *sposvxx*
DOUBLE PRECISION for *dposvxx*
COMPLEX for *cposvxx*
DOUBLE COMPLEX for *zposvxx*.

Array, size *ldx* by *.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:

$$\text{inv}(\text{diag}(s)) * X.$$

a Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

<i>af</i>	<p>If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>a</i> for the form of the equilibrated matrix.</p>
<i>b</i>	<p>If <i>equed</i> = 'N', <i>B</i> is not modified.</p> <p>If <i>equed</i> = 'Y', <i>B</i> is overwritten by $diag(s)*B$.</p>
<i>s</i>	<p>This array is an output argument if <i>fact</i>≠'F'. Each element of this array is a power of the radix. See the description of <i>s</i> in <i>Input Arguments</i> section.</p>
<i>rcond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.</p>
<i>rpvgrw</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Contains the reciprocal pivot growth factor:</p> $\ A\ /\ U\ $ <p>If this is much less than 1, the stability of the <i>LU</i> factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>X</i>, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i>.</p>
<i>berr</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>
<i>err_bnds_norm</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array of size <i>nrhs</i> by <i>n_err_bnds</i>. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:</p> <p>Normwise relative error in the <i>i</i>-th solution vector</p> $\frac{\max_j X_{true_{ji}} - X_{ji} }{\max_j X_{ji} }$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`equed` If `fact# 'F'`, then `equed` is an output argument. It specifies the form of equilibration that was done (see the description of `equed` in *Input Arguments* section).

`params` If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

`info` INTEGER. If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the i -th parameter had an illegal value.

If $0 < info \leq n$: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $params(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $err_bnds_norm(j,1) = 0.0$ or $err_bnds_comp(j,1) = 0.0$. See the definition of err_bnds_norm and err_bnds_comp for $err = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A and multiple right-hand sides.

Syntax

```
call sppsv( uplo, n, nrhs, ap, b, ldb, info )
call dppsv( uplo, n, nrhs, ap, b, ldb, info )
call cppsv( uplo, n, nrhs, ap, b, ldb, info )
call zppsv( uplo, n, nrhs, ap, b, ldb, info )
call ppsv( ap, b [,uplo] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

	If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>ap, b</i>	REAL for <i>sppsv</i> DOUBLE PRECISION for <i>dppsv</i> COMPLEX for <i>cppsv</i> DOUBLE COMPLEX for <i>zppsv</i> . Arrays: <i>ap</i> (size *), <i>b</i> (size <i>ldb</i> , *). The array <i>ap</i> contains the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>ap</i>	If <i>info</i> = 0, the upper or lower triangular part of <i>A</i> in packed storage is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *ppsv* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call sppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )
```

```
call dppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr,
berr, work, iwork, info )
```

```
call cppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

```
call zppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, ldx, rcond, ferr,
berr, work, rwork, info )
```

```
call ppsvx( ap, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{inv}(\text{diag}(s))*X = \text{diag}(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s)*A*\text{diag}(s)$ and B by $\text{diag}(s)*B$.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U^T*U \text{ (real)}, A = U^H*U \text{ (complex)}, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L*L^T \text{ (real)}, A = L*L^H \text{ (complex)}, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with *info* = i . Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, *info* = $n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F': on entry, <i>afp</i> contains the factored form of <i>A</i>. If <i>equed</i> = 'Y', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>s</i>. <i>ap</i> and <i>afp</i> will not be modified.</p> <p>If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>afp</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated if necessary, then copied to <i>afp</i> and factored.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>n</i>	<p>INTEGER. The order of matrix <i>A</i>; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides; the number of columns in <i>B</i>; $nrhs \geq 0$.</p>
<i>ap, afp, b, work</i>	<p>REAL for <code>sppsvx</code> DOUBLE PRECISION for <code>dppsvx</code> COMPLEX for <code>cppsvx</code> DOUBLE COMPLEX for <code>zppsvx</code>.</p> <p>Arrays: (size *), <i>afp</i>(size *), <i>b</i>(size <i>ldb</i> by *), <i>work</i>(*).</p> <p>The array <i>ap</i> contains the upper or lower triangle of the original symmetric/Hermitian matrix <i>A</i> in <i>packed storage</i> (see Matrix Storage Schemes). In case when <i>fact</i> = 'F' and <i>equed</i> = 'Y', <i>ap</i> must contain the equilibrated matrix $diag(s) * A * diag(s)$.</p> <p>The array <i>afp</i> is an input argument if <i>fact</i> = 'F' and contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of <i>A</i> in the same storage format as <i>A</i>. If <i>equed</i> is not 'N', then <i>afp</i> is the factored form of the equilibrated matrix <i>A</i>.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p> <p><i>work</i>(*) is a workspace array.</p> <p>The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; $ldb \geq \max(1, n)$.</p>

<i>equed</i>	<p>CHARACTER*1. Must be 'N' or 'Y'.</p> <p><i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done:</p> <p>if <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N');</p> <p>if <i>equed</i> = 'Y', equilibration was done, that is, <i>A</i> has been replaced by $diag(s)A*diag(s)$.</p>
<i>s</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i>. This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument.</p> <p>If <i>equed</i> = 'N', <i>s</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive.</p>
<i>ldx</i>	<p>INTEGER. The leading dimension of the output array <i>x</i>; $ldx \geq \max(1, n)$.</p>
<i>iwork</i>	<p>INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.</p>
<i>rwork</i>	<p>REAL for <i>cppsvx</i>;</p> <p>DOUBLE PRECISION for <i>zppsvx</i>.</p> <p>Workspace array, size at least $\max(1, n)$; used in complex flavors only.</p>

Output Parameters

<i>x</i>	<p>REAL for <i>sppsvx</i></p> <p>DOUBLE PRECISION for <i>dppsvx</i></p> <p>COMPLEX for <i>cppsvx</i></p> <p>DOUBLE COMPLEX for <i>zppsvx</i>.</p> <p>Array, size <i>ldx</i> by *.</p> <p>If <i>info</i> = 0 or <i>info</i> = <i>n</i>+1, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that if <i>equed</i> = 'Y', <i>A</i> and <i>B</i> are modified on exit, and the solution to the equilibrated system is $inv(diag(s))*X$. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.</p>
<i>ap</i>	<p>Array <i>ap</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'.</p> <p>If <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>ap</i> is overwritten by $diag(s)*A*diag(s)$.</p>

<i>afp</i>	<p>If <i>fact</i> = 'N' or 'E', then <i>afp</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ap</i> for the form of the equilibrated matrix.</p>
<i>b</i>	<p>Overwritten by $diag(s)*B$, if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.</p>
<i>s</i>	<p>This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.</p>
<i>rcond</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.</p>
<i>ferr</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i>-th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i>, and is almost always a slight overestimate of the true error.</p>
<i>berr</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>
<i>equed</i>	<p>If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).</p>
<i>info</i>	<p>INTEGER. If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, and $i \leq n$, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned.</p>

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ppsvx` interface are as follows:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>afp</i>	Holds the matrix AF of size $(n*(n+1)/2)$.
<i>s</i>	Holds the vector of length n . Default value for each element is $s(i) = 1.0_WP$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

See Also

Matrix Storage Schemes

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band coefficient matrix A and multiple right-hand sides.

Syntax

```
call spbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call pbsv( ab, b [,uplo] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A . The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals of the matrix A if $uplo = 'U'$, or the number of subdiagonals if $uplo = 'L'$; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ab, b</i>	REAL for <code>spbsv</code> DOUBLE PRECISION for <code>dpbsv</code> COMPLEX for <code>cpbsv</code> DOUBLE COMPLEX for <code>zpbsv</code> . Arrays: ab (size $ldab$ by *), b (size ldb by *). The array ab contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in <i>band storage</i> (see Matrix Storage Schemes). The second dimension of ab must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The leading dimension of the array ab ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>ab</i>	The upper or lower triangular part of A (in band storage) is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> , in the same storage format as A .
<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value. If <i>info</i> = i , the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbsv` interface are as follows:

<i>ab</i>	Holds the array A of size $(kd+1, n)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call spbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
call dpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
call cpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
call zpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
call pbsvx( ab, b, x [,uplo] [,afb] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?pbsvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$$diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(s)*A*diag(s)$ and B by $diag(s)*B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$$A = U^T*U \text{ (real), } A = U^H*U \text{ (complex), if } uplo = 'U',$$

$$\text{or } A = L*L^T \text{ (real), } A = L*L^H \text{ (complex), if } uplo = 'L',$$

where U is an upper triangular band matrix and L is a lower triangular band matrix.

3. If the leading i -by- i principal minor is not positive definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

<code>fact</code>	<p>CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If <code>fact = 'F'</code>: on entry, <code>afb</code> contains the factored form of A. If <code>equed = 'Y'</code>, the matrix A has been equilibrated with scaling factors given by s. <code>ab</code> and <code>afb</code> will not be modified.</p> <p>If <code>fact = 'N'</code>, the matrix A will be copied to <code>afb</code> and factored.</p> <p>If <code>fact = 'E'</code>, the matrix A will be equilibrated if necessary, then copied to <code>afb</code> and factored.</p>
<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If <code>uplo = 'U'</code>, the upper triangle of A is stored.</p> <p>If <code>uplo = 'L'</code>, the lower triangle of A is stored.</p>
<code>n</code>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>

kd INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

nrhs INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

ab, afb, b, work REAL for spbsvx
 DOUBLE PRECISION for dpbsvx
 COMPLEX for cpbsvx
 DOUBLE COMPLEX for zpbsvx.
 Arrays: *ab*(size *ldab* by *), *afb*(size *ldafb* by *), *b*(size *ldb* by *), *work*(*).
 The array *ab* contains the upper or lower triangle of the matrix *A* in *band storage* (see [Matrix Storage Schemes](#)).
 If *fact* = 'F' and *equed* = 'Y', then *ab* must contain the equilibrated matrix $diag(s) * A * diag(s)$. The second dimension of *ab* must be at least $\max(1, n)$.
 The array *afb* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A* in the same storage format as *A*. If *equed* = 'Y', then *afb* is the factored form of the equilibrated matrix *A*. The second dimension of *afb* must be at least $\max(1, n)$.
 The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.
work(*) is a workspace array.
 The dimension of *work* must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

ldab INTEGER. The leading dimension of *ab*; $ldab \geq kd+1$.

ldafb INTEGER. The leading dimension of *afb*; $ldafb \geq kd+1$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

equed CHARACTER*1. Must be 'N' or 'Y'.
equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:
 if *equed* = 'N', no equilibration was done (always true if *fact* = 'N')
 if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$.

s REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Array, size (n). The array s contains the scale factors for A . This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $equed = 'N'$, s is not accessed.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

ldx INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for `cpbsvx`
 DOUBLE PRECISION for `zpbsvx`.
 Workspace array, size at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x REAL for `spbsvx`
 DOUBLE PRECISION for `dpbsvx`
 COMPLEX for `cpbsvx`
 DOUBLE COMPLEX for `zpbsvx`.
 Array, size ldx by $*$.
 If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that if $equed = 'Y'$, A and B are modified on exit, and the solution to the equilibrated system is $inv(diag(s)) * X$. The second dimension of x must be at least $\max(1, nrhs)$.

ab On exit, if $fact = 'E'$ and $equed = 'Y'$, A is overwritten by $diag(s) * A * diag(s)$.

afb If $fact = 'N'$ or $'E'$, then *afb* is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix A (if $fact = 'N'$), or of the equilibrated matrix A (if $fact = 'E'$). See the description of *ab* for the form of the equilibrated matrix.

b Overwritten by $diag(s) * B$, if $equed = 'Y'$; not changed if $equed = 'N'$.

s This array is an output argument if $fact \neq 'F'$. See the description of s in *Input Arguments* section.

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

*info*INTEGER. If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbsvx` interface are as follows:

<i>ab</i>	Holds the array A of size $(kd+1, n)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.

<i>afb</i>	Holds the array <i>AF</i> of size $(kd+1, n)$.
<i>s</i>	Holds the vector with the number of elements <i>n</i> . Default value for each element is $s(i) = 1.0_WP$.
<i>ferr</i>	Holds the vector with the number of elements <i>nrhs</i> .
<i>berr</i>	Holds the vector with the number of elements <i>nrhs</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then <i>afb</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be 'N' or 'Y'. The default value is 'N'.

See Also

Matrix Storage Schemes

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
call sptsv( n, nrhs, d, e, b, ldb, info )
call dptsv( n, nrhs, d, e, b, ldb, info )
call cptsv( n, nrhs, d, e, b, ldb, info )
call zptsv( n, nrhs, d, e, b, ldb, info )
call ptsv( d, e, b [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the real or complex system of linear equations $A^*X = B$, where *A* is an *n*-by-*n* symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

A is factored as $A = L^*D^*L^T$ (real flavors) or $A = L^*D^*L^H$ (complex flavors), and the factored form of *A* is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>d</i>	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, dimension at least $\max(1, n)$. Contains the diagonal elements of the tridiagonal matrix A .

e, b

REAL for `sptsv`

DOUBLE PRECISION for `dptsv`

COMPLEX for `cptsv`

DOUBLE COMPLEX for `zptsv`.

Arrays: e (size $n - 1$), b (size ldb by $*$). The array e contains the $(n - 1)$ subdiagonal elements of A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

d

Overwritten by the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of A .

e

Overwritten by the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the factorization of A .

b

Overwritten by the solution matrix X .

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless $i = n$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ptsv` interface are as follows:

d

Holds the vector of length n .

e

Holds the vector of length $(n-1)$.

b

Holds the matrix B of size $(n, nrhs)$.

See Also

[Matrix Storage Schemes](#)

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call sptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, info )
```

```
call dptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, info )
```

```
call cptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

```
call zptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

```
call ptsvx( d, e, b, x [,df] [,ef] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the Cholesky factorization $A = L^*D^*L^T$ (real)/ $A = L^*D^*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A^*X = B$, where A is a n -by- n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If $fact = 'N'$, the matrix A is factored as $A = L^*D^*L^T$ (real flavors)/ $A = L^*D^*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T^*D^*U$ (real flavors)/ $A = U^H^*D^*U$ (complex flavors).
2. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact CHARACTER*1. Must be 'F' or 'N'.
Specifies whether or not the factored form of the matrix A is supplied on entry.
If $fact = 'F'$: on entry, df and ef contain the factored form of A . Arrays d , e , df , and ef will not be modified.
If $fact = 'N'$, the matrix A will be copied to df and ef , and factored.

<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>d, df, rwork</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>d</i> (size <i>n</i>), <i>df</i> (size <i>n</i>), <i>rwork</i> (<i>n</i>). The array <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>A</i> . The array <i>df</i> is an input argument if <i>fact</i> = 'F' and on entry contains the <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of <i>A</i> . The array <i>rwork</i> is a workspace array used for complex flavors only.
<i>e,ef,b,work</i>	REAL for sptsvx DOUBLE PRECISION for dptsvx COMPLEX for cptsvx DOUBLE COMPLEX for zptsvx. Arrays: <i>e</i> (size <i>n</i> - 1), <i>ef</i> (size <i>n</i> - 1), <i>b</i> (size <i>ldb</i> , *), <i>work</i> (*). The array <i>e</i> contains the (<i>n</i> - 1) subdiagonal elements of the tridiagonal matrix <i>A</i> . The array <i>ef</i> is an input argument if <i>fact</i> = 'F' and on entry contains the (<i>n</i> - 1) subdiagonal elements of the unit bidiagonal factor <i>L</i> from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of <i>A</i> . The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The array <i>work</i> is a workspace array. The dimension of <i>work</i> must be at least $2*n$ for real flavors, and at least <i>n</i> for complex flavors.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.

Output Parameters

<i>x</i>	REAL for sptsvx DOUBLE PRECISION for dptsvx COMPLEX for cptsvx DOUBLE COMPLEX for zptsvx. Array, size <i>ldx</i> by *. If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
----------	---

<i>df, ef</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>df, ef</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.</p>
<i>ferr</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i>-th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i>, and is almost always a slight overestimate of the true error.</p>
<i>berr</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, and $i \leq n$, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned.</p> <p>If <i>info</i> = <i>i</i>, and $i = n + 1$, then <i>U</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ptsvx` interface are as follows:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length (<i>n</i> -1).

<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>df</i>	Holds the vector of length n .
<i>ef</i>	Holds the vector of length $(n-1)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```
call ssysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call sysv( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.

<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a, b, work</i>	<p>REAL for <i>ssysv</i></p> <p>DOUBLE PRECISION for <i>dsysv</i></p> <p>COMPLEX for <i>csysv</i></p> <p>DOUBLE COMPLEX for <i>zsysv</i>.</p> <p>Arrays: <i>a</i>(size <i>lda</i> by *), <i>b</i>(size <i>ldb</i> by *), <i>work</i>(*).</p> <p>The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix <i>A</i> (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p> <p><i>work</i> is a workspace array, dimension at least $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; $lwork \geq 1$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>. See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by <code>?sytrf</code> .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i>, as determined by <code>?sytrf</code>.</p> <p>If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the <i>i</i>-th row and column of <i>A</i> was interchanged with the <i>k</i>-th row and column.</p> <p>If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i-1</i>, and (<i>i-1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p> <p>If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i>, and (<i>i+1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p>

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value. If <code>info = i</code> , d_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sysv` interface are as follows:

<code>a</code>	Holds the matrix <i>A</i> of size (n, n) .
<code>b</code>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length <i>n</i> .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using `lwork = n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

[Matrix Storage Schemes](#)

?sysv_rook

*Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix *A* and multiple right-hand sides.*

Syntax

```
call ssysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsysv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

```
call sysv_rook( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The `?sysv_rook` routine is called to compute the factorization of a complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.
<i>a, b, work</i>	REAL for <code>ssysv_rook</code> DOUBLE PRECISION for <code>dsysv_rook</code> COMPLEX for <code>csysv_rook</code> DOUBLE COMPLEX for <code>zsysv_rook</code> . Arrays: <i>a</i> (size <i>lda</i> by *), <i>b</i> (size <i>ldb</i> by *), <i>work</i> (*). The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

<i>a</i>	If $info = 0$, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by <code>sytrf_rook</code> .
<i>b</i>	If $info = 0$, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> , as determined by <code>sytrf_rook</code> . If $ipiv(k) > 0$, then rows and columns <i>k</i> and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block. If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block. If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.
<i>work(1)</i>	If $info = 0$, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$, d_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sysv_rook` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call ssysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, iwork, info )
```

```
call dsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, iwork, info )
```

```
call csysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, rwork, info )
```

```
call zsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, rwork, info )
```

```
call sysvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, where A is a n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

fact CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If $fact = 'F'$: on entry, *af* and *ipiv* contain the factored form of A . Arrays *a*, *af*, and *ipiv* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *af* and factored.

uplo CHARACTER*1. Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of *A* is stored:
If *uplo* = 'U', the upper triangle of *A* is stored.
If *uplo* = 'L', the lower triangle of *A* is stored.

n INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

a, af, b, work REAL for *ssysvx*
DOUBLE PRECISION for *dsysvx*
COMPLEX for *csysvx*
DOUBLE COMPLEX for *zsysvx*.
Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb* by *), *work*(*).
The array *a* contains the upper or the lower triangular part of the symmetric matrix *A* (see *uplo*). The second dimension of *a* must be at least $\max(1, n)$.
The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^T$ or $A = L * D * L^T$ as computed by ?[sytrf](#). The second dimension of *af* must be at least $\max(1, n)$.
The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.
work(*) is a workspace array, dimension at least $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.
Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by ?[sytrf](#).
If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.
If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of *A* was interchanged with the *m*-th row and column.
If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i> .
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for csysvx; DOUBLE PRECISION for zsysvx. Workspace array, size at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for ssysvx DOUBLE PRECISION for dsysvx COMPLEX for csysvx DOUBLE COMPLEX for zsysvx. Array, size <i>ldx</i> by *. If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
<i>af, ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>af, ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector <i>x</i> (<i>j</i>) (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to <i>x</i> (<i>j</i>), <i>ferr</i> (<i>j</i>) is an estimated upper bound for the magnitude of the largest element in (<i>x</i> (<i>j</i>) - <i>xtrue</i>) divided by the magnitude of the largest element in <i>x</i> (<i>j</i>). The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.

<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, then $d_{i,i}$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and $i = n + 1$, then D is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sysvx` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least $\max(1, m*n)$, where for real flavors $m = 3$ and for complex flavors $m = 2$. For better performance, try using $lwork = \max(1, m*n, n*blocksize)$, where *blocksize* is the optimal block size for `?sytrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?sysvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

```
call ssysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call dsysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
iwork, info )
```

```
call csysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zsysvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a real or complex system of linear equations $A*X = B$, where *A* is an *n*-by-*n* real symmetric/Hermitian matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where *eps* is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?sysvxx` performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and *B* by $\text{diag}(s) * B$.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

$$A = U * D * U^T, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L * D * L^T, \text{ if } \text{uplo} = 'L',$$

where *U* or *L* is a product of permutation and unit upper (lower) triangular matrices, and *D* is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i, i) = 0$, so that *D* is exactly singular, the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.
5. By default, unless *params*(1) is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by $\text{diag}(r)$ so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F', 'N', or 'E'. Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored. If <i>fact</i> = 'F', on entry, <i>af</i> and <i>ipiv</i> contain the factored form of <i>A</i> . If <i>equed</i> is not 'N', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>s</i> . Parameters <i>a</i> , <i>af</i> , and <i>ipiv</i> are not modified. If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>af</i> and factored. If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated, if necessary, copied to <i>af</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.

a, af, b, work

REAL for *ssysvxx*
 DOUBLE PRECISION for *dsysvxx*
 COMPLEX for *csysvxx*
 DOUBLE COMPLEX for *zsysvxx*.

Arrays: *a*(size *lda* by *), *af*(size *ldaf* by *), *b*(size *ldb*, *), *work*(*).

The array *a* contains the symmetric matrix *A* as specified by *uplo*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A* and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A* and the strictly upper triangular part of *a* is not referenced. The second dimension of *a* must be at least $\max(1, n)$.

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* and *L* from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ as computed by [?sytrf](#). The second dimension of *af* must be at least $\max(1, n)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array. The dimension of *work* must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of the array *a*; $lda \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of the array *af*; $ldaf \geq \max(1, n)$.

ipiv

INTEGER.

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D* as determined by [?sytrf](#). If *ipiv*(*k*) > 0, rows and columns *k* and *ipiv*(*k*) were interchanged and *D*(*k*,*k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*-1) < 0, rows and columns *k*-1 and -*ipiv*(*k*) were interchanged and *D*(*k*-1:*k*, *k*-1:*k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, rows and columns *k*+1 and -*ipiv*(*k*) were interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

equed

CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$.

s

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Array, size (n). The array s contains the scale factors for A . If $equed = 'Y'$, A is multiplied on the left and right by $diag(s)$.

This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

<i>ldb</i>	INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in the <i>Output Arguments</i> section below.
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size <i>nparams</i> . Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument. <i>params</i> (1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors). =0.0 No refinement is performed and no error bounds are computed. =1.0 Use the extra-precise refinement algorithm. (Other values are reserved for future use.) <i>params</i> (2) : Maximum number of residual computations allowed for refinement. Default 10.0 Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for *ssysvxx*
 DOUBLE PRECISION for *dsysvxx*
 COMPLEX for *csysvxx*
 DOUBLE COMPLEX for *zsysvxx*.

Array, size *ldx* by *nrhs*).

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:

$$\text{inv}(\text{diag}(s)) * X.$$

a If *fact* = 'E' and *equed* = 'Y', overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

af If *fact* = 'N', *af* is an output argument and on exit returns the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^T$ or $A = L * D * L^T$.

b If *equed* = 'N', *B* is not modified.
 If *equed* = 'Y', *B* is overwritten by $\text{diag}(s) * B$.

s This array is an output argument if *fact* ≠ 'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Contains the reciprocal pivot growth factor:

$\|A\|/\|U\|$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. Up to three pieces of information are returned.

The first index in *err_bnds_norm(i, :)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:, err)* contains the following three fields:

- | | |
|---------------|---|
| <i>err</i> =1 | "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. |
| <i>err</i> =2 | "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true. |
| <i>err</i> =3 | Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors |

and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first (`(:, n_err_bnds)`) entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

ipiv

If *fact* = 'N', *ipiv* is an output argument and on exit contains details of the interchanges and the block structure D , as determined by `ssytrf` for single precision flavors and `dsytrf` for double precision flavors.

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = $-i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = $n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

```
call chesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

```
call zhesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

```
call hesv( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array a stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$. If <i>uplo</i> = 'L', the array a stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>a, b, work</i>	COMPLEX for chesv DOUBLE COMPLEX for zhesv. Arrays: a (size <i>lda</i> by *), bb (size <i>ldb</i> by *), $work$ (*). The array a contains the upper or the lower triangular part of the Hermitian matrix A (see <i>uplo</i>). The second dimension of a must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$. $work$ is a workspace array, dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the $work$ array ($lwork \geq 1$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

<i>a</i>	If $info = 0$, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by <code>?hetrf</code> .
<i>b</i>	If $info = 0$, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> , as determined by <code>?hetrf</code> . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the <i>i</i> -th row and column of <i>A</i> was interchanged with the <i>k</i> -th row and column. If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i-1</i> , and (<i>i-1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column. If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i> , and (<i>i+1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column.
<i>work</i> (1)	If $info = 0$, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$, d_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hesv` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?hesv_rook

Computes the solution to the system of linear equations for Hermitian matrices using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call chesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call hesv_rook( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for *X* the complex system of linear equations $A * X = B$, where *A* is an *n*-by-*n* Hermitian matrix, and *X* and *B* are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman ("rook") diagonal pivoting method is used to factor *A* as

$A = U * D * U^H$ if *uplo* = 'U', or

$A = L * D * L^H$ if *uplo* = 'L',

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

[hetrf_rook](#) is called to compute the factorization of a complex Hermitian matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of *A* is then used to solve the system of equations $A * X = B$ by calling ?HETRS_ROOK, which uses BLAS level 2 routines.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of <i>A</i> is stored:

If `uplo = 'U'`, the array `a` stores the upper triangular part of the matrix `A`.

If `uplo = 'L'`, the array `a` stores the lower triangular part of the matrix `A`.

`n` INTEGER. The number of linear equations, which is the order of matrix `A`; $n \geq 0$.

`nrhs` INTEGER. The number of right-hand sides, the number of columns in `B`; $nrhs \geq 0$.

`a, b, work` COMPLEX for `chesv_rook`
COMPLEX*16 for `zhesv_rook`.

Arrays: `a`(size `lda` by `*`), `b`(size `ldb` by `*`), `work`(`*`).

The array `a` contains the Hermitian matrix `A`. If `uplo = 'U'`, the leading `n`-by-`n` upper triangular part of `a` contains the upper triangular part of the matrix `A`, and the strictly lower triangular part of `a` is not referenced. If `uplo = 'L'`, the leading `n`-by-`n` lower triangular part of `a` contains the lower triangular part of the matrix `A`, and the strictly upper triangular part of `a` is not referenced. The second dimension of `a` must be at least $\max(1, n)$.

The array `b` contains the `n`-by-`nrhs` right hand side matrix `B`. The second dimension of `b` must be at least $\max(1, nrhs)$.

`work` is a workspace array, dimension at least $\max(1, lwork)$.

`lda` INTEGER. The leading dimension of `a`; $lda \geq \max(1, n)$.

`ldb` INTEGER. The leading dimension of `b`; $ldb \geq \max(1, n)$.

`lwork` INTEGER. The size of the `work` array ($lwork \geq 1$).

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of `lwork`.

Output Parameters

`a` If `info = 0`, `a` is overwritten by the block-diagonal matrix `D` and the multipliers used to obtain the factor `U` (or `L`) from the factorization of `A` as computed by [hetrf_rook](#).

`b` If `info = 0`, `b` is overwritten by the `n`-by-`nrhs` solution matrix `X`.

`ipiv` INTEGER.

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of `D`, as determined by [?hetrf_rook](#).

- If `uplo = 'U'`:

If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

- If $uplo = 'L'$:

If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

work(1)

If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, D_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hesv_rook` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also Matrix Storage Schemes

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian coefficient matrix A , and provides error bounds on the solution.

Syntax

```
call chesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
```

```
call zhesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr, berr, work, lwork, rwork, info )
```

```
call hesvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A*X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hesvx` performs the following steps:

1. If `fact = 'N'`, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<code>fact</code>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <code>fact = 'F'</code> : on entry, <code>af</code> and <code>ipiv</code> contain the factored form of A . Arrays <code>a</code> , <code>af</code> , and <code>ipiv</code> are not modified. If <code>fact = 'N'</code> , the matrix A is copied to <code>af</code> and factored.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the upper triangular part of the Hermitian matrix A , and A is factored as $U*D*U^H$. If <code>uplo = 'L'</code> , the array <code>a</code> stores the lower triangular part of the Hermitian matrix A ; A is factored as $L*D*L^H$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<code>a, af, b, work</code>	COMPLEX for <code>chesvx</code> DOUBLE COMPLEX for <code>zhesvx</code> . Arrays: <code>a</code> (size <code>lda</code> by *), <code>af</code> (size <code>ldaf</code> by *), <code>b</code> (size <code>ldb</code> by *), <code>work</code> (*) .

The array *a* contains the upper or the lower triangular part of the Hermitian matrix *A* (see *uplo*). The second dimension of *a* must be at least $\max(1, n)$.

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by [?hetrf](#). The second dimension of *af* must be at least $\max(1, n)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work(*) is a workspace array of dimension at least $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by [?hetrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i-1*, and (*i-1*)-th row and column of *A* was interchanged with the *m*-th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and (*i+1*)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

lwork INTEGER. The size of the *work* array.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of *lwork*.

rwork REAL for `chesvx`
 DOUBLE PRECISION for `zhesvx`.
 Workspace array, size at least $\max(1, n)$.

Output Parameters

x COMPLEX for `chesvx`
 DOUBLE COMPLEX for `zhesvx`.

Array, size ldx by $*$.

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the system of equations. The second dimension of x must be at least $\max(1, nrhs)$.

af, ipiv

These arrays are output arguments if $fact = 'N'$. See the description of *af, ipiv* in *Input Arguments* section.

rcond

REAL for *chesvx*

DOUBLE PRECISION for *zhesvx*.

An estimate of the reciprocal condition number of the matrix A . If *rcond* is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for *chesvx*

DOUBLE PRECISION for *zhesvx*.

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for *chesvx*

DOUBLE PRECISION for *zhesvx*.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

work(1)

If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hesvx` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>af</i>	Holds the matrix <i>AF</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

Application Notes

The value of *lwork* must be at least $2*n$. For better performance, try using $lwork = n*blocksize$, where *blocksize* is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

See Also

Matrix Storage Schemes

?hesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a Hermitian indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

```
call chesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

```
call zhesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a complex/double complex system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?hesvxx` performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U * D * U^T, \text{ if } \text{uplo} = \text{'U'},$$

$$\text{or } A = L * D * L^T, \text{ if } \text{uplo} = \text{'L'},$$

where U or L is a product of permutation and unit upper (lower) triangular matrices, and D is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i, i) = 0$, so that D is exactly singular, the routine returns with *info* = i . Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless *params*(1) is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(r)$ so that it solves the original system before equilibration.

Input Parameters

fact CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$, on entry, af and $ipiv$ contain the factored form of A . If $equed$ is not $'N'$, the matrix A has been equilibrated with scaling factors given by s . Parameters a , af , and $ipiv$ are not modified.

If $fact = 'N'$, the matrix A will be copied to af and factored.

If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.

$uplo$

CHARACTER*1. Must be $'U'$ or $'L'$.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

If $uplo = 'L'$, the lower triangle of A is stored.

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

$nrhs$

INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

$a, af, b, work$

COMPLEX for `chesvxx`

DOUBLE COMPLEX for `zhesvxx`.

Arrays: a (size lda by $*$), af (size $ldaf$ by $*$), $b(ldb, *)$, $work(*)$.

The array a contains the Hermitian matrix A as specified by $uplo$. If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced. The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U and L from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ as computed by `?hetrf`. The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 5*n)$.

lda

INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

$ldaf$

INTEGER. The leading dimension of the array af ; $ldaf \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, size at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains details of the interchanges and the block structure of D as determined by `?sytrf`.

If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

equed

CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

if $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $diag(s)*A*diag(s)$.

s

REAL for *chesvxx*

DOUBLE PRECISION for *zhesvxx*.

Array, size (n). The array s contains the scale factors for A . If $equed = 'Y'$, A is multiplied on the left and right by $diag(s)$.

This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default 10

Aggressive Set to 100 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

params(3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

rwork

REAL for *chesvxx*

DOUBLE PRECISION for *zhesvxx*.

Workspace array, size at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x

COMPLEX for *chesvxx*

DOUBLE COMPLEX for *zhesvxx*.

Array, size *ldx* by *nrhs*.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:

$$\text{inv}(\text{diag}(s)) * X.$$

a

If *fact* = 'E' and *equed* = 'Y', overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

af

If *fact* = 'N', *af* is an output argument and on exit returns the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^T$ or $A = L * D * L^T$.

b

If *equed* = 'N', *B* is not modified.

If *equed* = 'Y', *B* is overwritten by $\text{diag}(s) * B$.

s

This array is an output argument if *fact* ≠ 'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond

REAL for *chesvxx*

DOUBLE PRECISION for zhesvxx.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading $info$ columns of A .

berr

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array of size $nrhs$ by n_err_bnds . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in $err_bnds_norm(i, :)$ corresponds to the i -th right-hand side.

The second index in $err_bnds_norm(:, err)$ contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for chesvxx and $\sqrt{n} * dlamch(\epsilon)$ for zhesvxx.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `chesvxx` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zhesvxx`. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `chesvxx` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zhesvxx` to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for `chesvxx`

DOUBLE PRECISION for `zhesvxx`.

Array of size `nrhs` by `n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first `(:,n_err_bnds)` entries are returned.

The first index in `err_bnds_comp(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `chesvxx` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zhesvxx`.

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for *chesvxx* and $\sqrt{n} * \text{dlamch}(\epsilon)$ for *zhesvxx*. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for *chesvxx* and $\sqrt{n} * \text{dlamch}(\epsilon)$ for *zhesvxx* to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where *x* is the solution for the current right-hand side and *s* scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of *z* are approximately 1.

ipiv If *fact* = 'N', *ipiv* is an output argument and on exit contains details of the interchanges and the block structure *D*, as determined by *ssytrf* for single precision flavors and *dsytrf* for double precision flavors.

equed If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$. See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also
Matrix Storage Schemes

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
call sspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call cspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call spsv( ap, b [,uplo] [,ipiv] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap, b</i>	REAL for sspsv DOUBLE PRECISION for dspsv COMPLEX for cspsv DOUBLE COMPLEX for zspsv. Arrays: <i>ap</i> (size *), <i>b</i> (size <i>ldb</i> by *). The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in packed storage (see Matrix Storage Schemes).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ap The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by `?sptrf`, stored as a packed triangular matrix in the same storage format as A .

b If $info = 0$, b is overwritten by the solution matrix X .

$ipiv$ INTEGER.

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by `?sptrf`.

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$info$ INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spsv` interface are as follows:

ap Holds the array A of size $(n*(n+1)/2)$.

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector with the number of elements n .

$uplo$ Must be 'U' or 'L'. The default value is 'U'.

See Also

Matrix Storage Schemes

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call sspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, iwork, info )
call dspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, iwork, info )
call cspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
call zspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
call spsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, where A is a n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If $fact = 'F'$: on entry, <i>afp</i> and <i>ipiv</i> contain the factored form of A . Arrays <i>ap</i> , <i>afp</i> , and <i>ipiv</i> are not modified. If $fact = 'N'$, the matrix A is copied to <i>afp</i> and factored.
-------------	---

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *ap* stores the upper triangular part of the symmetric matrix A , and A is factored as $U^*D^*U^T$.

If *uplo* = 'L', the array *ap* stores the lower triangular part of the symmetric matrix A ; A is factored as $L^*D^*L^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

ap, afp, b, work REAL for `sspsvx`
DOUBLE PRECISION for `dspsvx`
COMPLEX for `cspsvx`
DOUBLE COMPLEX for `zspsvx`.

Arrays: *ap*(size *), *afp*(size *), *b*(size *ldb* by *), *work*(*).

The array *ap* contains the upper or lower triangle of the symmetric matrix A in *packed storage* (see [Matrix Storage Schemes](#)).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ as computed by `?sptrf`, in the same storage format as A .

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n+1)/2)$; the second dimension of *b* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv INTEGER.

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of D , as determined by `?sptrf`.

If *ipiv*(*i*) = $k > 0$, then $d_{i,i}$ is a 1-by-1 block, and the *i*-th row and column of A was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = $-m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of A was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = $-m < 0$, then D has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of A was interchanged with the *m*-th row and column.

<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for <i>cspsvx</i> DOUBLE PRECISION for <i>zspsvx</i> . Workspace array, size at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for <i>sspsvx</i> DOUBLE PRECISION for <i>dspsvx</i> COMPLEX for <i>cspsvx</i> DOUBLE COMPLEX for <i>zspsvx</i> . Array, size <i>ldx</i> by *. If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
<i>afp, ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>afp, ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix <i>D</i> is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and $i = n + 1$, then <i>D</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are

computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spsvx` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>afp</i>	Holds the array <i>AF</i> of size $(n*(n+1)/2)$.
<i>ipiv</i>	Holds the vector with the number of elements <i>n</i> .
<i>ferr</i>	Holds the vector with the number of elements <i>nrhs</i> .
<i>berr</i>	Holds the vector with the number of elements <i>nrhs</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

?hpsv

*Computes the solution to the system of linear equations with a Hermitian coefficient matrix *A* stored in packed format, and multiple right-hand sides.*

Syntax

```
call chpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call hpsv( ap, b [,uplo] [,ipiv] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves for *X* the system of linear equations $A*X = B$, where *A* is an *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.</p>
<i>n</i>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides; the number of columns in B; $nrhs \geq 0$.</p>
<i>ap, b</i>	<p>COMPLEX for <code>chpsv</code> DOUBLE COMPLEX for <code>zhpsv</code>.</p> <p>Arrays: <i>ap</i>(size *), <i>b</i>(size <i>ldb</i> by *).</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor U or L, as specified by <i>uplo</i>, in <i>packed storage</i> (see Matrix Storage Schemes).</p> <p>The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; $ldb \geq \max(1, n)$.</p>

Output Parameters

<i>ap</i>	<p>The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by <code>?hptrf</code>, stored as a packed triangular matrix in the same storage format as A.</p>
<i>b</i>	<p>If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix X.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D, as determined by <code>?hptrf</code>.</p> <p>If <i>ipiv</i>(<i>i</i>) = $k > 0$, then d_{ii} is a 1-by-1 block, and the <i>i</i>-th row and column of A was interchanged with the <i>k</i>-th row and column.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>(<i>i</i>) = <i>ipiv</i>(<i>i</i>-1) = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>-1, and (<i>i</i>-1)-th row and column of A was interchanged with the <i>m</i>-th row and column.</p> <p>If <i>uplo</i> = 'L' and <i>ipiv</i>(<i>i</i>) = <i>ipiv</i>(<i>i</i>+1) = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>+1)-th row and column of A was interchanged with the <i>m</i>-th row and column.</p>

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpsv` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

See Also

[Matrix Storage Schemes](#)

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
call chpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
call zhpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
call hpsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A*X = B$, where *A* is a *n*-by-*n* Hermitian matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hpsvx` performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U*D*U^H$ or $A = L*D*L^H$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>fact</i>	<p>CHARACTER*1. Must be 'F' or 'N'.</p> <p>Specifies whether or not the factored form of the matrix A has been supplied on entry.</p> <p>If <i>fact</i> = 'F': on entry, <i>afp</i> and <i>ipiv</i> contain the factored form of A. Arrays <i>ap</i>, <i>afp</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix A is copied to <i>afp</i> and factored.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored and how A is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the Hermitian matrix A, and A is factored as $U^*D^*U^H$.</p> <p>If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the Hermitian matrix A, and A is factored as $L^*D^*L^H$.</p>
<i>n</i>	<p>INTEGER. The order of matrix A; $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.</p>
<i>ap, afp, b, work</i>	<p>COMPLEX for <code>chpsvx</code></p> <p>DOUBLE COMPLEX for <code>zhpsvx</code>.</p> <p>Arrays: <i>ap</i>(size *), <i>afp</i>(size *), <i>b</i>(size <i>ldb</i> by *), <i>work</i>(*).</p> <p>The array <i>ap</i> contains the upper or lower triangle of the Hermitian matrix A in <i>packed storage</i> (see Matrix Storage Schemes).</p> <p>The array <i>afp</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by <code>?hptrf</code>, in the same storage format as A.</p> <p>The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.</p> <p><i>work</i>(*) is a workspace array.</p> <p>The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 2*n)$.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of b; $ldb \geq \max(1, n)$.</p>
<i>ipiv</i>	<p>INTEGER.</p>

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by `?hptrf`.

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and $ipiv(i) = ipiv(i-1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i-1*, and (*i-1*)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and $ipiv(i) = ipiv(i+1) = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and (*i+1*)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

rwork REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
Workspace array, size at least $\max(1, n)$.

Output Parameters

x COMPLEX for `chpsvx`
DOUBLE COMPLEX for `zhpsvx`.
Array, size *ldx* by *.
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least $\max(1, nrhs)$.

afp, *ipiv* These arrays are output arguments if *fact* = 'N'. See the description of *afp*, *ipiv* in *Input Arguments* section.

rcond REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr REAL for `chpsvx`
DOUBLE PRECISION for `zhpsvx`.
Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to $x(j)$, *ferr*(*j*) is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

<i>berr</i>	<p>REAL for <code>chpsvx</code></p> <p>DOUBLE PRECISION for <code>zhpsvx</code>.</p> <p>Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.</p>
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.</p> <p>If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpsvx` interface are as follows:

<i>ap</i>	Holds the array A of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>afp</i>	Holds the array AF of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector with the number of elements n .
<i>ferr</i>	Holds the vector with the number of elements $nrhs$.
<i>berr</i>	Holds the vector with the number of elements $nrhs$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If $fact = 'F'$, then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

See Also

Matrix Storage Schemes

Least Squares and Eigenvalue Problem Routines

This section includes descriptions of LAPACK [computational routines](#) and [driver routines](#) for solving linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. For a full reference on LAPACK routines and related information see [LUG].

Least Squares Problems. A typical *least squares problem* is as follows: given a matrix A and a vector b , find the vector x that minimizes the sum of squares $\sum_i ((Ax)_i - b_i)^2$ or, equivalently, find the vector x that minimizes the 2-norm $\|Ax - b\|_2$.

In the most usual case, A is an m -by- n matrix with $m \geq n$ and $\text{rank}(A) = n$. This problem is also referred to as finding the *least squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the *QR* factorization of the matrix A (see [QR Factorization](#)).

If $m < n$ and $\text{rank}(A) = m$, there exist an infinite number of solutions x which exactly satisfy $Ax = b$, and thus minimize the norm $\|Ax - b\|_2$. In this case it is often useful to find the unique solution that minimizes $\|x\|_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the *LQ* factorization of the matrix A (see [LQ Factorization](#)).

In the general case you may have a *rank-deficient least squares problem*, with $\text{rank}(A) < \min(m, n)$: find the *minimum-norm least squares solution* that minimizes both $\|x\|_2$ and $\|Ax - b\|_2$. In this case (or when the rank of A is in doubt) you can use the *QR* factorization with pivoting or *singular value decomposition* (see [Singular Value Decomposition](#)).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

If A is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric* eigenvalue problem. Routines for solving this type of problems are described in the section [Symmetric Eigenvalue Problems](#).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section [Nonsymmetric Eigenvalue Problems](#).

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors x that satisfy one of the following equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section [Generalized Symmetric-Definite Eigenvalue Problems](#).

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in "LAPACK Routines: Linear Equations" as well as with BLAS routines described in "BLAS and Sparse BLAS Routines".

For example, to solve a set of least squares problems minimizing $\|Ax - b\|_2^2$ for all columns b of a given matrix B (where A and B are real matrices), you can call `?gexrf` to form the factorization $A = QR$, then call `?ormqr` to compute $C = Q^H B$ and finally call the BLAS routine `?trsm` to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine `?gels` can be used.

Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

Orthogonal Factorizations

Singular Value Decomposition

Symmetric Eigenvalue Problems

Generalized Symmetric-Definite Eigenvalue Problems

Nonsymmetric Eigenvalue Problems

Generalized Nonsymmetric Eigenvalue Problems

Generalized Singular Value Decomposition

See also the respective [driver routines](#).

Orthogonal Factorizations

This section describes the LAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included.

QR Factorization. Assume that A is an m -by- n matrix to be factored.

If $m \geq n$, the QR factorization is given by

$$A = \begin{pmatrix} Q_1 & & \\ & R & \\ & & 0 \end{pmatrix} \begin{pmatrix} I_n \\ & & 0 \end{pmatrix}$$

where R is an n -by- n upper triangular matrix with real diagonal elements, and Q is an m -by- m orthogonal (or unitary) matrix.

You can use the QR factorization for solving the following least squares problem: minimize $\|Ax - b\|^2$ where A is a full-rank m -by- n matrix ($m \geq n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the QR factorization is given by

$$A = QR = Q(R_1 R_2)$$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m -by- n matrix A is as follows. If $m \leq n$,

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = (LQ_1)$$

where L is an m -by- m lower triangular matrix with real diagonal elements, and Q is an n -by- n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 & & \\ & L_2 & \\ & & 0 \end{pmatrix} Q$$

where L_1 is an n -by- n lower triangular matrix, L_2 is rectangular, and Q is an n -by- n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m -by- n matrix of rank m ($m < n$). After factoring the matrix, compute the solution vector x as follows: solve $Ly = b$ for y , and then compute $x = (Q_1)^H y$.

Table "Computational Routines for Orthogonal Factorization" lists LAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	geqrf	geqpf	orgqr	ormqr
	geqrfp	geqp3	ungqr	unmqr
general matrices, blocked QR factorization	geqrt			gemqrt
general matrices, RQ factorization	gerqf		orgrq	ormrq
			ungrq	unmrq
general matrices, LQ factorization	gelqf		orglq	ormlq
			unglq	unmlq
general matrices, QL factorization	geqlf		orgql	ormql
			ungql	unmql
trapezoidal matrices, RZ factorization	tzzrf			ormrz
				unmrz
pair of matrices, generalized QR factorization	ggqrf			
pair of matrices, generalized RQ factorization	ggrqf			
triangular-pentagonal matrices, blocked QR factorization	tpqrt			tpmqrt

[?geqrf](#)

Computes the QR factorization of a general m -by- n matrix.

Syntax

```
call sgeqrf(m, n, a, lda, tau, work, lwork, info)
call dgeqrf(m, n, a, lda, tau, work, lwork, info)
call cgeqrf(m, n, a, lda, tau, work, lwork, info)
call zgeqrf(m, n, a, lda, tau, work, lwork, info)
call geqrf(a [, tau] [, info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for `sgeqrf`
 DOUBLE PRECISION for `dgeqrf`
 COMPLEX for `cgeqrf`
 DOUBLE COMPLEX for `zgeqrf`.

Arrays: *a*(*lda*,*) contains the matrix A . The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See [Application Notes](#) for the suggested value of *lwork*.

Output Parameters

a Overwritten by the factorization data as follows:
 The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, present the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors (see [Orthogonal Factorizations](#)).

tau REAL for `sgeqrf`
 DOUBLE PRECISION for `dgeqrf`
 COMPLEX for `cgeqrf`
 DOUBLE COMPLEX for `zgeqrf`.

Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors (see [Orthogonal Factorizations](#)).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *geqrf* interface are the following:

a Holds the matrix *A* of size (*m*,*n*).
tau Holds the vector of length min(*m*,*n*)

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

?geqrf (this routine) to factorize $A = QR$;
 ormqr to compute $C = Q^T * B$ (for real matrices);
 unmqr to compute $C = Q^H * B$ (for complex matrices);
 trsm (a BLAS routine) to solve $R * X = C$.

(The columns of the computed X are the least squares solution vectors x .)

To compute the elements of Q explicitly, call

`orgqr` (for real matrices)

`ungqr` (for complex matrices).

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?geqrfp

Computes the QR factorization of a general m -by- n matrix with non-negative diagonal elements.

Syntax

```
call sgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

```
call dgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

```
call cgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

```
call zgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

Include Files

- `mkl.fi`

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed. The diagonal entries of R are real and nonnegative.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for `sgeqrfp`
 DOUBLE PRECISION for `dgeqrfp`
 COMPLEX for `cgeqrfp`
 DOUBLE COMPLEX for `zgeqrfp`.

Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , present the orthogonal matrix <i>Q</i> as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations). The diagonal elements of the matrix <i>R</i> are real and non-negative.
<i>tau</i>	REAL for <i>sgeqrfp</i> DOUBLE PRECISION for <i>dgeqrfp</i> COMPLEX for <i>cgeqrfp</i> DOUBLE COMPLEX for <i>zgeqrfp</i> . Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix <i>Q</i> in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *geqrfp* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length $\min(m, n)$

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

<code>?geqrfp</code> (this routine)	to factorize $A = QR$;
<code>ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R * X = C$.

(The columns of the computed *X* are the least squares solution vectors *x*.)

To compute the elements of *Q* explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?geqrt

Computes a blocked QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
call sgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call dgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call cgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call zgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call geqrt(a, t, nb[, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower triangular part of array a .

NOTE

The 1s along the diagonal of V are not stored in a .

Let $k = \min(m, n)$. The number of blocks is $b = \text{ceiling}(k/nb)$, where each block is of order nb except for the last block, which is of order $ib = k - (b-1)*nb$. For each of the b blocks, a upper triangular block reflector factor is computed: t_1, t_2, \dots, t_b . The nb -by- nb (and ib -by- ib for the last block) t s are stored in the nb -by- n array t as

$$t = (t_1 t_2 \dots t_b).$$

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
nb	INTEGER. The block size to be used in the blocked QR ($\min(m, n) \geq nb \geq 1$).
$a, work$	REAL for <code>sgeqrt</code> DOUBLE PRECISION for <code>dgeqrt</code> COMPLEX for <code>cgeqrt</code> COMPLEX*16 for <code>zgeqrt</code> . Arrays: <code>a</code> DIMENSION (lda, n) contains the m -by- n matrix A . <code>work</code> DIMENSION (nb, n) is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least nb .

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array t , present the orthogonal matrix Q as a product of $\min(m,n)$ elementary reflectors (see Orthogonal Factorizations).
<i>t</i>	REAL for sgeqrt DOUBLE PRECISION for dgeqrt COMPLEX for cgeqrt COMPLEX*16 for zgeqrt. Array, DIMENSION (ldt , $\min(m, n)$). The upper triangular block reflector's factors stored as a sequence of upper triangular blocks.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info < 0$ and $info = -i$, the i th argument had an illegal value.

?gemqrt

Multiplies a general matrix by the orthogonal/unitary matrix Q of the QR factorization formed by ?geqrt.

Syntax

```
call sgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call dgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call cgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call zgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call gemqrt( v, t, c, k, nb[, trans][, side][, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?gemqrt routine overwrites the general real or complex m -by- n matrix C with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * C$	$C * Q$
<i>trans</i> = 'T':	$Q^T * C$	$C * Q^T$
<i>trans</i> = 'C':	$Q^H * C$	$C * Q^H$

where Q is a real orthogonal (complex unitary) matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k) = I - V * T * V^T$ for real flavors, and

$Q = H(1) H(2) \dots H(k) = I - V * T * V^H$ for complex flavors,

generated using the compact WY representation as returned by [geqrt](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	<p>CHARACTER</p> <p>= 'L': apply Q, Q^T, or Q^H from the left.</p> <p>= 'R': apply Q, Q^T, or Q^H from the right.</p>
<i>trans</i>	<p>CHARACTER</p> <p>= 'N', no transpose, apply Q.</p> <p>= 'T', transpose, apply Q^T.</p> <p>= 'C', transpose, apply Q^H.</p>
<i>m</i>	INTEGER. The number of rows in the matrix C , ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix C , ($n \geq 0$).
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>nb</i>	<p>INTEGER.</p> <p>The block size used for the storage of t, $k \geq nb \geq 1$. This must be the same value of <i>nb</i> used to generate t in geqrt.</p>
<i>v</i>	<p>REAL for sgemqrt</p> <p>DOUBLE PRECISION for dgemqrt</p> <p>COMPLEX for cgemqrt</p> <p>COMPLEX*16 for zgemqrt.</p> <p>Array, DIMENSION (ldv, k).</p> <p>The ith column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by geqrt in the first k columns of its array argument a.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array v.</p> <p>if <i>side</i> = 'L', ldv must be at least $\max(1, m)$;</p> <p>if <i>side</i> = 'R', ldv must be at least $\max(1, n)$.</p>
<i>t</i>	<p>REAL for sgemqrt</p> <p>DOUBLE PRECISION for dgemqrt</p> <p>COMPLEX for cgemqrt</p> <p>COMPLEX*16 for zgemqrt.</p> <p>Array, DIMENSION (ldt, k).</p> <p>The upper triangular factors of the block reflectors as returned by geqrt.</p>
<i>ldt</i>	INTEGER. The leading dimension of the array t . ldt must be at least nb .

<i>c</i>	<p>REAL for <code>sgemqrt</code></p> <p>DOUBLE PRECISION for <code>dgemqrt</code></p> <p>COMPLEX for <code>cgemqrt</code></p> <p>COMPLEX*16 for <code>zgemqrt</code>.</p> <p>The <i>m</i>-by-<i>n</i> matrix <i>C</i>.</p>
<i>ldc</i>	<p>INTEGER. The leading dimension of the array <i>c</i>. <i>ldc</i> must be at least $\max(1, m)$.</p>
<i>work</i>	<p>REAL for <code>sgemqrt</code></p> <p>DOUBLE PRECISION for <code>dgemqrt</code></p> <p>COMPLEX for <code>cgemqrt</code></p> <p>COMPLEX*16 for <code>zgemqrt</code>.</p> <p>Workspace array.</p> <p>If <i>side</i> = 'L' DIMENSION <i>n</i>*<i>nb</i>.</p> <p>If <i>side</i> = 'R' DIMENSION <i>m</i>*<i>nb</i>.</p>

Output Parameters

<i>c</i>	<p>Overwritten by the product Q^*C, C^*Q, Q^T*C, C^*Q^T, Q^H*C, or C^*Q^H as specified by <i>side</i> and <i>trans</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>= 0: the execution is successful.</p> <p>< 0: if <i>info</i> = -<i>i</i>, the <i>i</i>th argument had an illegal value.</p>

?geqpf

Computes the QR factorization of a general *m*-by-*n* matrix with pivoting.

Syntax

```
call sgeqpf(m, n, a, lda, jpvt, tau, work, info)
call dgeqpf(m, n, a, lda, jpvt, tau, work, info)
call cgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call zgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call geqpf(a, jpvt [,tau] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine is deprecated and has been replaced by routine [geqp3](#).

The routine `?geqpf` forms the QR factorization of a general *m*-by-*n* matrix *A* with column pivoting: $A^*P = Q^*R$ (see [Orthogonal Factorizations](#)). Here *P* denotes an *n*-by-*n* permutation matrix.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgeqpf</code> DOUBLE PRECISION for <code>dgeqpf</code> COMPLEX for <code>cgeqpf</code> DOUBLE COMPLEX for <code>zgeqpf</code> . Arrays: a ($lda, *$) contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ ($lwork$) is a workspace array. The size of the $work$ array must be at least $\max(1, 3*n)$ for real flavors and at least $\max(1, n)$ for complex flavors.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$jpvt$	INTEGER. Array, size at least $\max(1, n)$. On entry, if $jpvt(i) > 0$, the i -th column of A is moved to the beginning of $A*P$ before the computation, and fixed in place during the computation. If $jpvt(i) = 0$, the i th column of A is a free column (that is, it may be interchanged during the computation with any other free column).
$rwork$	REAL for <code>cgeqpf</code> DOUBLE PRECISION for <code>zgeqpf</code> . A workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau , present the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations).
tau	REAL for <code>sgeqpf</code> DOUBLE PRECISION for <code>dgeqpf</code> COMPLEX for <code>cgeqpf</code> DOUBLE COMPLEX for <code>zgeqpf</code> . Array, size at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .
$jpvt$	Overwritten by details of the permutation matrix P in the factorization $A*P = Q*R$. More precisely, the columns of $A*P$ are the columns of A in the following order:

$jpvt(1), jpvt(2), \dots, jpvt(n).$

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `geqpf` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>jpvt</i>	Holds the vector of length n .
<i>tau</i>	Holds the vector of length $\min(m,n)$

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqpf</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array *jpvt* specifies the permutation order.)

To compute the elements of Q explicitly, call

`orgqr` (for real matrices)

`ungqr` (for complex matrices).

?geqp3

Computes the QR factorization of a general m -by- n matrix with column pivoting using level 3 BLAS.

Syntax

```
call sgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call dgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call cgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call zgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call geqp3(a, jpvt [,tau] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the QR factorization of a general m -by- n matrix A with column pivoting: $A*P = Q*R$ (see [Orthogonal Factorizations](#)) using Level 3 BLAS. Here P denotes an n -by- n permutation matrix. Use this routine instead of [geqpf](#) for better performance.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgeqp3</code> DOUBLE PRECISION for <code>dgeqp3</code> COMPLEX for <code>cgeqp3</code> DOUBLE COMPLEX for <code>zgeqp3</code> . Arrays: a ($lda, *$) contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; must be at least $\max(1, 3*n+1)$ for real flavors, and at least $\max(1, n+1)$ for complex flavors. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> below for details.
$jpvt$	INTEGER. Array, size at least $\max(1, n)$.

On entry, if $jpvt(i) \neq 0$, the i -th column of A is moved to the beginning of AP before the computation, and fixed in place during the computation.

If $jpvt(i) = 0$, the i -th column of A is a free column (that is, it may be interchanged during the computation with any other free column).

rwork

REAL for `cgeqp3`

DOUBLE PRECISION for `zgeqp3`.

A workspace array, size at least $\max(1, 2*n)$. Used in complex flavors only.

Output Parameters

a

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, present the orthogonal matrix Q as a product of $\min(m,n)$ elementary reflectors (see [Orthogonal Factorizations](#)).

tau

REAL for `sgeqp3`

DOUBLE PRECISION for `dgeqp3`

COMPLEX for `cgeqp3`

DOUBLE COMPLEX for `zgeqp3`.

Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .

jpvt

Overwritten by details of the permutation matrix P in the factorization $A*P = Q*R$. More precisely, the columns of AP are the columns of A in the following order:

$jpvt(1), jpvt(2), \dots, jpvt(n)$.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `geqp3` interface are the following:

a Holds the matrix A of size (m,n) .

jpvt Holds the vector of length n .

tau Holds the vector of length $\min(m,n)$

Application Notes

To solve a set of least squares problems minimizing $\|A*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqp3</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R * X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array $jpvt$ specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

`?orgqr`

Generates the real orthogonal matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
call sorgqr(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
call orgqr(a, tau [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine generates the whole or part of m -by- m orthogonal matrix Q of the QR factorization formed by the routines `geqrf` or `geqpf`. Use this routine after a call to `sgeqrf/dgeqrf` or `sgeqpf/dgeqpf`.

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call?orgqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call?orgqr(m, p, p, a, lda, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A :

```
call?orgqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
call?orgqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the orthogonal matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed ($0 \leq n \leq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
$a, \tau, work$	REAL for sorgqr DOUBLE PRECISION for dorgqr Arrays: $a(lda,*)$ and $\tau(*)$ are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf. The second dimension of a must be at least $\max(1, n)$. The size of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array ($lwork \geq n$). If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

a	Overwritten by n leading columns of the m -by- m orthogonal matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orgqr` interface are the following:

<code>a</code>	Holds the matrix A of size (m,n) .
<code>tau</code>	Holds the vector of length (k)

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly orthogonal matrix by a matrix E such that

$$\|E\|_2 = O(\varepsilon) \|A\|_2 \text{ where } \varepsilon \text{ is the machine precision.}$$

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If $n = k$, the number is approximately $(2/3) * n^2 * (3m - n)$.

The complex counterpart of this routine is `ungqr`.

?ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf or ?geqpf.

Syntax

```
call sormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call dormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call ormqr(a, tau, c [,side] [,trans] [,info])
```

Include Files

- `mkl.fi, lapack.f90`

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QR factorization formed by the routines `geqrf` or `geqpf`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q * C$, $Q^T * C$, $C * Q$, or $C * Q^T$ (overwriting the result on C).

Input Parameters

<i>side</i>	<p>CHARACTER*1. Must be either 'L' or 'R'.</p> <p>If <i>side</i>='L', Q or Q^T is applied to C from the left.</p> <p>If <i>side</i>='R', Q or Q^T is applied to C from the right.</p>
<i>trans</i>	<p>CHARACTER*1. Must be either 'N' or 'T'.</p> <p>If <i>trans</i>='N', the routine multiplies C by Q.</p> <p>If <i>trans</i>='T', the routine multiplies C by Q^T.</p>
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>$0 \leq k \leq m$ if <i>side</i>='L';</p> <p>$0 \leq k \leq n$ if <i>side</i>='R'.</p>
<i>a, tau, c, work</i>	<p>REAL for <i>sgeqrf</i></p> <p>DOUBLE PRECISION for <i>dgeqrf</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> and <i>tau(*)</i> are the arrays returned by <i>sgeqrf</i> / <i>dgeqrf</i> or <i>sgeqpf</i> / <i>dgeqpf</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, k)$.</p> <p>The size of <i>tau</i> must be at least $\max(1, k)$.</p> <p><i>c ldc,*)</i> contains the matrix C.</p> <p>The second dimension of <i>c</i> must be at least $\max(1, n)$</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of <i>a</i>. Constraints:</p> <p>if <i>side</i> = 'L', $lda \geq \max(1, m)$;</p> <p>if <i>side</i> = 'R', $lda \geq \max(1, n)$.</p>
<i>ldc</i>	<p>INTEGER. The leading dimension of <i>c</i>. Constraint:</p> <p>$ldc \geq \max(1, m)$.</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array. Constraints:</p> <p>$lwork \geq \max(1, n)$ if <i>side</i> = 'L';</p> <p>$lwork \geq \max(1, m)$ if <i>side</i> = 'R'.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormqr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>k</i>). $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n \cdot blocksize$ (if *side* = 'L') or $lwork = m \cdot blocksize$ (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is `unmqr`.

?ungqr

Generates the complex unitary matrix *Q* of the QR factorization formed by `?geqrf`.

Syntax

```
call cunqqr(m, n, k, a, lda, tau, work, lwork, info)
```

```
call zungqr(m, n, k, a, lda, tau, work, lwork, info)
```

```
call unqqr(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of m -by- m unitary matrix Q of the QR factorization formed by the routines `geqrf` or `geqpf`. Use this routine after a call to `cgeqrf/zgeqrf` or `cgeqpf/zgeqpf`.

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call?unqqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call?unqqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A :

```
call?unqqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
call?unqqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The order of the unitary matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of Q to be computed ($0 \leq n \leq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
$a, \tau, work$	COMPLEX for <code>cunqqr</code> DOUBLE COMPLEX for <code>zungqr</code> Arrays: $a(lda,*)$ and $\tau(*)$ are the arrays returned by <code>cgeqrf/zgeqrf</code> or <code>cgeqpf/zgeqpf</code> . The second dimension of a must be at least $\max(1, n)$. The size of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	Overwritten by n leading columns of the m -by- m unitary matrix Q .
<i>work</i> (1)	If $info = 0$, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ungqr` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>tau</i>	Holds the vector of length (k) .

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$.

If $n = k$, the number is approximately $(8/3) * n^2 * (3m - n)$.

The real counterpart of this routine is [orgqr](#).

?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

```
call cunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmqr(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a rectangular complex matrix C by Q or Q^H , where Q is the unitary matrix Q of the QR factorization formed by the routines [geqrf](#) or [geqpf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, c, tau, work</i>	COMPLEX for cgeqrf DOUBLE COMPLEX for zgeqrf. Arrays: <i>a(lda,*)</i> and <i>tau(*)</i> are the arrays returned by cgeqrf / zgeqrf or cgeqpf / zgeqpf. The second dimension of <i>a</i> must be at least $\max(1, k)$. The size of <i>tau</i> must be at least $\max(1, k)$. <i>c(ldc,*)</i> contains the m -by- n matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*. Constraints:

$lda \geq \max(1, m)$ if *side* = 'L';

$lda \geq \max(1, n)$ if *side* = 'R'.

ldc INTEGER. The leading dimension of *c*. Constraint:

$ldc \geq \max(1, m)$.

lwork INTEGER. The size of the *work* array. Constraints:

$lwork \geq \max(1, n)$ if *side* = 'L';

$lwork \geq \max(1, m)$ if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmqr` interface are the following:

a Holds the matrix *A* of size (*r*,*k*).

$r = m$ if *side* = 'L'.

$r = n$ if *side* = 'R'.

tau Holds the vector of length (*k*).

c Holds the matrix *C* of size (*m*,*n*).

side Must be 'L' or 'R'. The default value is 'L'.

trans Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormqr](#).

?gelqf

Computes the LQ factorization of a general m -by- n matrix.

Syntax

```
call sgelqf(m, n, a, lda, tau, work, lwork, info)
call dgelqf(m, n, a, lda, tau, work, lwork, info)
call cgelqf(m, n, a, lda, tau, work, lwork, info)
call zgelqf(m, n, a, lda, tau, work, lwork, info)
call gelqf(a [, tau] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the LQ factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgelqf</code> DOUBLE PRECISION for <code>dgelqf</code>

COMPLEX for `cgelqf`

DOUBLE COMPLEX for `zgelqf`.

Arrays:

Array $a(lda,*)$ contains the matrix A .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array; at least $\max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the factorization data as follows:

The elements on and below the diagonal of the array contain the m -by- $\min(m,n)$ lower trapezoidal matrix R (R is lower triangular if $m \leq n$); the elements above the diagonal, with the array tau , represent the orthogonal matrix Q as a product of elementary reflectors.

tau REAL for `sgelqf`
DOUBLE PRECISION for `dgelqf`
COMPLEX for `cgelqf`
DOUBLE COMPLEX for `zgelqf`.

Array, size at least $\max(1, \min(m, n))$.

Contains scalars that define elementary reflectors for the matrix Q (see [Orthogonal Factorizations](#)).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gelqf` interface are the following:

a Holds the matrix A of size (m,n) .

tau Holds the vector of length $\min(m,n)$.

Application Notes

For better performance, try using $lwork = m * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least squares problem minimizing $\|A*x - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

`?gelqf` (this routine) to factorize $A = L*Q$;

`trsm` (a BLAS routine) to solve $L*Y = B$ for *Y*;

`ormlq` to compute $X = (Q_1)^T * Y$ (for real matrices);

`unmlq` to compute $X = (Q_1)^H * Y$ (for complex matrices).

(The columns of the computed *X* are the minimum-norm solution vectors *x*. Here *A* is an *m*-by-*n* matrix with $m < n$; Q_1 denotes the first *m* columns of *Q*).

To compute the elements of *Q* explicitly, call

`orglq` (for real matrices)

`unglq` (for complex matrices).

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call sorglq(m, n, k, a, lda, tau, work, lwork, info)
```

```
call dorglq(m, n, k, a, lda, tau, work, lwork, info)
```

```
call orglq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of n -by- n orthogonal matrix Q of the LQ factorization formed by the routines [gelqf](#). Use this routine after a call to [sgelqf/dgelqf](#).

Usually Q is determined from the LQ factorization of an p -by- n matrix A with $n \geq p$. To compute the whole matrix Q , use:

```
call ?orglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
call ?orglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of the leading k rows of A , use:

```
call ?orglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of A , use:

```
call ?orgqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

m	INTEGER. The number of rows of Q to be computed ($0 \leq m \leq n$).
n	INTEGER. The order of the orthogonal matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
$a, \tau, work$	REAL for sorglq DOUBLE PRECISION for dorglq Arrays: $a(lda,*)$ and $\tau(*)$ are the arrays returned by sgelqf/dgelqf . The second dimension of a must be at least $\max(1, n)$. The size of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array; at least $\max(1, m)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by *m* leading rows of the *n*-by-*n* orthogonal matrix *Q*.
work(1) If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.
info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orglq` interface are the following:

a Holds the matrix *A* of size (*m*,*n*).
tau Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* = *m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If *m* = *k*, the number is approximately $(2/3) * m^2 * (3n - m)$.

The complex counterpart of this routine is [unglq](#).

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call sormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call orm1q(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the LQ factorization formed by the routine [gelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, c, tau, work</i>	REAL for sormlq DOUBLE PRECISION for dormlq. Arrays: <i>a(lda,*)</i> and <i>tau(*)</i> are arrays returned by ?gelqf. The second dimension of <i>a</i> must be: at least $\max(1, m)$ if <i>side</i> = 'L'; at least $\max(1, n)$ if <i>side</i> = 'R'. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c(ldc,*)</i> contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$
 $work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by $side$ and $trans$).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormlq` interface are the following:

a Holds the matrix A of size (k,m) .

tau Holds the vector of length (k) .

c Holds the matrix C of size (m,n) .

$side$ Must be 'L' or 'R'. The default value is 'L'.

$trans$ Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ (if $side = 'L'$) or $lwork = m*blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork*= -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmlq](#).

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
call cunglq(m, n, k, a, lda, tau, work, lwork, info)
call zunglq(m, n, k, a, lda, tau, work, lwork, info)
call unglq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of *n*-by-*n* unitary matrix *Q* of the *LQ* factorization formed by the routines [gelqf](#). Use this routine after a call to [cgelqf/zgelqf](#).

Usually *Q* is determined from the *LQ* factorization of an *p*-by-*n* matrix *A* with $n < p$. To compute the whole matrix *Q*, use:

```
call ?unglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading *p* rows of *Q*, which form an orthonormal basis in the space spanned by the rows of *A*, use:

```
call ?unglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *LQ* factorization of the leading *k* rows of *A*, use:

```
call ?unglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading *k* rows of Q^k , which form an orthonormal basis in the space spanned by the leading *k* rows of *A*, use:

```
call ?ungqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

<i>m</i>	INTEGER. The number of rows of <i>Q</i> to be computed ($0 \leq m \leq n$).
<i>n</i>	INTEGER. The order of the unitary matrix <i>Q</i> ($n \geq m$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($0 \leq k \leq m$).
<i>a</i> , <i>tau</i> , <i>work</i>	COMPLEX for cunglq

DOUBLE COMPLEX for `zunglq`

Arrays: $a(lda,*)$ and $tau(*)$ are the arrays returned by `cgelqf/zgelqf`.

The second dimension of a must be at least $\max(1, n)$.

The dimension of tau must be at least $\max(1, k)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array; at least $\max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by m leading rows of the n -by- n unitary matrix Q .

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unglq` interface are the following:

a Holds the matrix A of size (m,n) .

tau Holds the vector of length (k) .

Application Notes

For better performance, try using $lwork = m*blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m+n)*k^2 + (16/3)*k^3$.

If $m = k$, the number is approximately $(8/3)*m^2*(3n - m)$.

The real counterpart of this routine is [orglq](#).

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by [?gelqf](#).

Syntax

```
call cunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call zunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call unmlq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a real m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the LQ factorization formed by the routine [gelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, c, tau, work</i>	COMPLEX for <code>cunmlq</code> DOUBLE COMPLEX for <code>zunmlq</code> . Arrays:

$a(lda,*)$ and $tau(*)$ are arrays returned by `?gelqf`.

The second dimension of a must be:

at least $\max(1, m)$ if $side = 'L'$;

at least $\max(1, n)$ if $side = 'R'$.

The size of tau must be at least $\max(1, k)$.

$c ldc, *$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (as specified by $side$ and $trans$).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmlq` interface are the following:

a Holds the matrix A of size (k, m) .

tau Holds the vector of length (k) .

c Holds the matrix C of size (m, n) .

$side$ Must be 'L' or 'R'. The default value is 'L'.

trans Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormlq](#).

?geqlf

Computes the QL factorization of a general m -by- n matrix.

Syntax

```
call sgeqlf(m, n, a, lda, tau, work, lwork, info)
call dgeqlf(m, n, a, lda, tau, work, lwork, info)
call cgeqlf(m, n, a, lda, tau, work, lwork, info)
call zgeqlf(m, n, a, lda, tau, work, lwork, info)
call geqlf(a [, tau] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the QL factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, *work* REAL for `sgeqlf`
 DOUBLE PRECISION for `dgeqlf`
 COMPLEX for `cgeqlf`
 DOUBLE COMPLEX for `zgeqlf`.

Arrays:
 Array *a*(*lda*,*) contains the matrix *A*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array; at least $\max(1, n)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten on exit by the factorization data as follows:
 if $m \geq n$, the lower triangle of the subarray *a*(*m-n+1:m*, 1:*n*) contains the *n*-by-*n* lower triangular matrix *L*; if $m \leq n$, the elements on and below the (*n-m*)-th superdiagonal contain the *m*-by-*n* lower trapezoidal matrix *L*; in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.

tau REAL for `sgeqlf`
 DOUBLE PRECISION for `dgeqlf`
 COMPLEX for `cgeqlf`
 DOUBLE COMPLEX for `zgeqlf`.
 Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix *Q* (see [Orthogonal Factorizations](#)).

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `geqlf` interface are the following:

<code>a</code>	Holds the matrix A of size (m,n) .
<code>tau</code>	Holds the vector of length $\min(m,n)$.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

<code>orgql</code>	to generate matrix Q (for real matrices);
<code>ungql</code>	to generate matrix Q (for complex matrices);
<code>ormql</code>	to apply matrix Q (for real matrices);
<code>unmql</code>	to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?orgql

Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
call sorgql(m, n, k, a, lda, tau, work, lwork, info)
call dorgql(m, n, k, a, lda, tau, work, lwork, info)
call orgql(a, tau [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates an m -by- n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf](#). Use this routine after a call to [sgeqlf/dgeqlf](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	REAL for <code>sorgql</code> DOUBLE PRECISION for <code>dorgql</code> Arrays: $a(lda,*)$, $tau(*)$. On entry, the $(n - k + i)$ th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>sgeqlf/dgeqlf</code> in the last k columns of its array argument a ; $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>sgeqlf/dgeqlf</code> ; The second dimension of a must be at least $\max(1, n)$. The size of tau must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the $work$ array; at least $\max(1, n)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

<i>a</i>	Overwritten by the last n columns of the m -by- m orthogonal matrix Q .
<i>work</i> (1)	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orgql` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n) .
----------	---

tau Holds the vector of length (*k*).

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [ungql](#).

?ungql

Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
call cungql(m, n, k, a, lda, tau, work, lwork, info)
call zungql(m, n, k, a, lda, tau, work, lwork, info)
call ungql(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates an *m*-by-*n* complex matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors $H(i)$ of order *m*: $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf/geqlf](#). Use this routine after a call to [cgeqlf/zgeqlf](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>Q</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix <i>Q</i> ($m \geq n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($n \geq k \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	COMPLEX for cungql DOUBLE COMPLEX for zungql Arrays: $a(lda,*)$, $tau(*)$, $work(lwork)$.

On entry, the $(n - k + i)$ th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `cgeqlf/zgeqlf` in the last k columns of its array argument a ;

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `cgeqlf/zgeqlf`;

The second dimension of a must be at least $\max(1, n)$.

The size of τ must be at least $\max(1, k)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array; at least $\max(1, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the last n columns of the m -by- m unitary matrix Q .

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ungql` interface are the following:

a Holds the matrix A of size (m, n) .

τ Holds the vector of length (k) .

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [orgql](#).

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
call sormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call dormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

```
call ormql(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QL factorization formed by the routine [geqlf](#).

Depending on the parameters *side* and *trans*, the routine [ormql](#) can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (overwriting the result over C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a, tau, c, work</i>	REAL for sormql DOUBLE PRECISION for dormql. Arrays: $a(lda,*)$, $tau(*)$, $c(ldc,*)$.

On entry, the i th column of a must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by `sgeqlf/dgeqlf` in the last k columns of its array argument a .

The second dimension of a must be at least $\max(1, k)$.

$\tau(i)$ must contain the scalar factor of the elementary reflector H_i , as returned by `sgeqlf/dgeqlf`.

The size of τ must be at least $\max(1, k)$.

$c(ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ;

if $side = 'L'$, $lda \geq \max(1, m)$;

if $side = 'R'$, $lda \geq \max(1, n)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by $side$ and $trans$).

work(1) If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormql` interface are the following:

a Holds the matrix A of size (r,k) .

$r = m$ if $side = 'L'$.

	$r = n$ if $side = 'R'$.
tau	Holds the vector of length (k).
c	Holds the matrix C of size (m,n).
$side$	Must be 'L' or 'R'. The default value is 'L'.
$trans$	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ (if $side = 'L'$) or $lwork = m*blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmql](#).

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by [?geqlf](#).

Syntax

```
call cunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmql(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the QL factorization formed by the routine [geqlf](#).

Depending on the parameters $side$ and $trans$, the routine [unmql](#) can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result over C).

Input Parameters

$side$	CHARACTER*1. Must be either 'L' or 'R'. If $side = 'L'$, Q or Q^H is applied to C from the left.
--------	--

If $side = 'R'$, Q or Q^H is applied to C from the right.

trans CHARACTER*1. Must be either 'N' or 'C'.
 If $trans = 'N'$, the routine multiplies C by Q .
 If $trans = 'C'$, the routine multiplies C by Q^H .

m INTEGER. The number of rows in the matrix C ($m \geq 0$).

n INTEGER. The number of columns in C ($n \geq 0$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:
 $0 \leq k \leq m$ if $side = 'L'$;
 $0 \leq k \leq n$ if $side = 'R'$.

a, tau, c, work COMPLEX for cunmql
 DOUBLE COMPLEX for zunmql.
 Arrays: $a(lda,*)$, $tau(*)$, $c ldc,*)$, $work(lwork)$.
 On entry, the i -th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `cgeqlf/zgeqlf` in the last k columns of its array argument a .
 The second dimension of a must be at least $\max(1, k)$.
 $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `cgeqlf/zgeqlf`.
 The size of tau must be at least $\max(1, k)$.
 $c(ldc,*)$ contains the m -by- n matrix C .
 The second dimension of c must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

lwork INTEGER. The size of the $work$ array. Constraints:
 $lwork \geq \max(1, n)$ if $side = 'L'$;
 $lwork \geq \max(1, m)$ if $side = 'R'$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
 See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product $Q^H C$, $Q^H C$, $C^H Q$, or $C^H Q^H$ (as specified by $side$ and $trans$).

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmql` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>k</i>). <i>r</i> = <i>m</i> if <code>side = 'L'</code> . <i>r</i> = <i>n</i> if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (<i>k</i>).
<code>c</code>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormql](#).

?gerqf

Computes the RQ factorization of a general m-by-n matrix.

Syntax

```
call sgerqf(m, n, a, lda, tau, work, lwork, info)
call dgerqf(m, n, a, lda, tau, work, lwork, info)
call cgerqf(m, n, a, lda, tau, work, lwork, info)
```

```
call zgerqf(m, n, a, lda, tau, work, lwork, info)
```

```
call gerqf(a [, tau] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the *RQ* factorization of a general *m*-by-*n* matrix *A* (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i> , <i>work</i>	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Arrays: Array <i>a</i> (<i>lda</i> ,*) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq \max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten on exit by the factorization data as follows: if $m \leq n$, the upper triangle of the subarray <i>a</i> (1: <i>m</i> , <i>n</i> - <i>m</i> +1: <i>n</i>) contains the <i>m</i> -by- <i>m</i> upper triangular matrix <i>R</i> ;
----------	--

if $m \geq n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ;

in both cases, the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.

τ	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Array, size at least $\max(1, \min(m, n))$. (See Orthogonal Factorizations.) Contains scalar factors of the elementary reflectors for the matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gerqf` interface are the following:

a	Holds the matrix A of size (m, n) .
τ	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using $lwork = m * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

<code>orgqr</code>	to generate matrix Q (for real matrices);
--------------------	---

`ungrq` to generate matrix Q (for complex matrices);

`ormrq` to apply matrix Q (for real matrices);

`unmrq` to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)
[Matrix Storage Schemes](#)

?orgrq

Generates the real matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
call sorgrq(m, n, k, a, lda, tau, work, lwork, info)
call dorgrq(m, n, k, a, lda, tau, work, lwork, info)
call orgrq(a, tau [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates an m -by- n real matrix with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the routines [gerqf](#). Use this routine after a call to [sgerqf/dgerqf](#).

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
$a, \tau, work$	REAL for <code>sorgrq</code> DOUBLE PRECISION for <code>dorgrq</code> Arrays: $a(lda,*)$, $\tau(*)$. On entry, the $(m - k + i)$ -th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>sgerqf/dgerqf</code> in the last k rows of its array argument a ; $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>sgerqf/dgerqf</code> ; The second dimension of a must be at least $\max(1, n)$. The size of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; at least $\max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the last <i>m</i> rows of the <i>n</i> -by- <i>n</i> orthogonal matrix <i>Q</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orgrq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).

Application Notes

For better performance, try using $lwork = m * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [ungrq](#).

?ungrq

Generates the complex matrix *Q* of the *RQ* factorization formed by [?gerqf](#).

Syntax

```
call cungrq(m, n, k, a, lda, tau, work, lwork, info)
```

```
call zungrq(m, n, k, a, lda, tau, work, lwork, info)
```

```
call ungrq(a, tau [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates an m -by- n complex matrix with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)^{H*} H(2)^{H*} \dots H(k)^H$ as returned by the routines [gerqf](#). Use this routine after a call to [cgerqf/zgerqf](#).

Input Parameters

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
$a, \tau, work$	<p>REAL for cungrq DOUBLE PRECISION for zungrq Arrays: $a(lda,*)$, $\tau(*)$, $work(lwork)$.</p> <p>On entry, the $(m - k + i)$th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgerqf/zgerqf in the last k rows of its array argument a; $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgerqf/zgerqf; The second dimension of a must be at least $\max(1, n)$. The size of τ must be at least $\max(1, k)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$lwork$	<p>INTEGER. The size of the $work$ array; at least $\max(1, m)$.</p> <p>If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of $lwork$.</p>

Output Parameters

a Overwritten by the m last rows of the n -by- n unitary matrix Q .

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ungrq` interface are the following:

<code>a</code>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<code>tau</code>	Holds the vector of length (<i>k</i>).

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [orgrq](#).

?ormrq

*Multiplies a real matrix by the orthogonal matrix *Q* of the RQ factorization formed by ?gerqf.*

Syntax

```
call sormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormrq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a real *m*-by-*n* matrix *C* by *Q* or *Q*^T, where *Q* is the real orthogonal matrix defined as a product of *k* elementary reflectors *H_i* : $Q = H_1 H_2 \dots H_k$ as returned by the RQ factorization routine [gerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (overwriting the result over *C*).

Input Parameters

<i>side</i>	<p>CHARACTER*1. Must be either 'L' or 'R'.</p> <p>If <i>side</i> = 'L', Q or Q^T is applied to C from the left.</p> <p>If <i>side</i> = 'R', Q or Q^T is applied to C from the right.</p>
<i>trans</i>	<p>CHARACTER*1. Must be either 'N' or 'T'.</p> <p>If <i>trans</i> = 'N', the routine multiplies C by Q.</p> <p>If <i>trans</i> = 'T', the routine multiplies C by Q^T.</p>
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>$0 \leq k \leq m$, if <i>side</i> = 'L';</p> <p>$0 \leq k \leq n$, if <i>side</i> = 'R'.</p>
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	<p>REAL for <code>sormrq</code></p> <p>DOUBLE PRECISION for <code>dormrq</code>.</p> <p>Arrays: <i>a</i>(<i>lda</i>,*), <i>tau</i>(*), <i>c</i>(<i>ldc</i>,*).</p> <p>On entry, the <i>i</i>th row of <i>a</i> must contain the vector which defines the elementary reflector H_i, for $i = 1, 2, \dots, k$, as returned by <code>sgerqf/dgerqf</code> in the last <i>k</i> rows of its array argument <i>a</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L', and at least $\max(1, n)$ if <i>side</i> = 'R'.</p> <p><i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector H_i, as returned by <code>sgerqf/dgerqf</code>.</p> <p>The size of <i>tau</i> must be at least $\max(1, k)$.</p> <p><i>c</i>(<i>ldc</i>,*) contains the <i>m</i>-by-<i>n</i> matrix C.</p> <p>The second dimension of <i>c</i> must be at least $\max(1, n)$</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, k)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array. Constraints:</p> <p>$lwork \geq \max(1, n)$ if <i>side</i> = 'L';</p> <p>$lwork \geq \max(1, m)$ if <i>side</i> = 'R'.</p>

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormrq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>m</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ (if *side* = 'L') or $lwork = m*blocksize$ (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmrq](#).

?unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
call cunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmrq(a, tau, c [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)^{H*} H(2)^{H*} \dots H(k)^{H*}$ as returned by the RQ factorization routine [gerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result over C).

Input Parameters

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>a, tau, c, work</i>	COMPLEX for cunmrq DOUBLE COMPLEX for zunmrq. Arrays: $a(lda,*)$, $tau(*)$, $c(ldc,*)$, $work(lwork)$. On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgerqf/zgerqf in the last k rows of its array argument a . The second dimension of a must be at least $\max(1, m)$ if <i>side</i> = 'L', and at least $\max(1, n)$ if <i>side</i> = 'R'.

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `cgerqf/zgerqf`.

The size of τ must be at least $\max(1, k)$.

$c(ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by $side$ and $trans$).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmrq` interface are the following:

a Holds the matrix A of size (k, m) .

τ Holds the vector of length (k) .

c Holds the matrix C of size (m, n) .

$side$ Must be 'L' or 'R'. The default value is 'L'.

$trans$ Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n * blocksize$ (if $side = 'L'$) or $lwork = m * blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormrq](#).

?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call stzrzf(m, n, a, lda, tau, work, lwork, info)
call dtzrzf(m, n, a, lda, tau, work, lwork, info)
call ctzrzf(m, n, a, lda, tau, work, lwork, info)
call ztzrzf(m, n, a, lda, tau, work, lwork, info)
call tzrzf(a [, tau] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix A to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix $A = [A1 A2] = [A_{1:m, 1:m}, A_{1:m, m+1:n}]$ is factored as

$$A = [R0] * Z,$$

where Z is an n -by- n orthogonal/unitary matrix, R is an m -by- m upper triangular matrix, and 0 is the m -by- $(n-m)$ zero matrix.

See [larz](#) that applies an elementary reflector returned by `?tzrzf` to a general matrix.

The `?tzrzf` routine replaces the deprecated `?tzrqf` routine.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq m$).
$a, work$	REAL for <code>stzrzf</code>

DOUBLE PRECISION for dtzrzf

COMPLEX for ctzrzf

DOUBLE COMPLEX for ztzrzf.

Arrays: $a(lda,*)$, $work(lwork)$.

The leading m -by- n upper trapezoidal part of the array a contains the matrix A to be factorized.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array;

$lwork \geq \max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten on exit by the factorization data as follows:

the leading m -by- m upper triangular part of a contains the upper triangular matrix R , and elements $m + 1$ to n of the first m rows of a , with the array tau , represent the orthogonal matrix Z as a product of m elementary reflectors.

tau REAL for stzrzf
DOUBLE PRECISION for dtzrzf
COMPLEX for ctzrzf
DOUBLE COMPLEX for ztzrzf.

Array, size at least $\max(1, m)$. Contains scalar factors of the elementary reflectors for the matrix Z .

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tzrzf` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>m</i>).

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - \tau u u^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - \tau u u^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z*(*k*) is an *l*-element vector. *tau* and *z*(*k*) are chosen to annihilate the elements of the *k*-th row of *A*.

The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u*(*k*) in the *k*-th row of *A*, such that the elements of *z*(*k*) are stored in *a*(*k*, *m*+1), ..., *a*(*k*, *n*).

The elements of *R* are returned in the upper triangular part of *A*.

The matrix *Z* is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

For better performance, try using *lwork* = *m***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

- `ormrz` to apply matrix Q (for real matrices)
`unmrz` to apply matrix Q (for complex matrices).

`?ormrz`

Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by `?tzzrf`.

Syntax

```
call sormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call dormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call ormz(a, tau, c, l [, side] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The `?ormrz` routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the factorization routine `tzzrf`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q * C$, $Q^T * C$, $C * Q$, or $C * Q^T$ (overwriting the result over C).

The matrix Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

The `?ormrz` routine replaces the deprecated `?latzm` routine.

Input Parameters

<code>side</code>	CHARACTER*1. Must be either 'L' or 'R'. If <code>side = 'L'</code> , Q or Q^T is applied to C from the left. If <code>side = 'R'</code> , Q or Q^T is applied to C from the right.
<code>trans</code>	CHARACTER*1. Must be either 'N' or 'T'. If <code>trans = 'N'</code> , the routine multiplies C by Q . If <code>trans = 'T'</code> , the routine multiplies C by Q^T .
<code>m</code>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<code>n</code>	INTEGER. The number of columns in C ($n \geq 0$).
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <code>side = 'L'</code> ; $0 \leq k \leq n$, if <code>side = 'R'</code> .
<code>l</code>	INTEGER.

The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints:

$0 \leq l \leq m$, if $side = 'L'$;

$0 \leq l \leq n$, if $side = 'R'$.

$a, tau, c, work$

REAL for `sormrz`

DOUBLE PRECISION for `dormrz`.

Arrays: $a(lda,*)$, $tau(*)$, $c ldc,*)$.

On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `stzrzf/dtzrzf` in the last k rows of its array argument a .

The second dimension of a must be at least $\max(1, m)$ if $side = 'L'$, and at least $\max(1, n)$ if $side = 'R'$.

$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `stzrzf/dtzrzf`.

The size of tau must be at least $\max(1, k)$.

$c ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc

INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$

INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c

Overwritten by the product Q^*C , $Q^T * C$, C^*Q , or C^*Q^T (as specified by $side$ and $trans$).

$work(1)$

If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormrz` interface are the following:

<code>a</code>	Holds the matrix A of size (k,m) .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>trans</code>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmrz](#).

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by [?tzrzf](#).

Syntax

```
call cunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call zunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call unmrz(a, tau, c, l [,side] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix defined as a product of k elementary reflectors $H(i)$:

$Q = H(1)^{H*} H(2)^{H*} \dots H(k)^H$ as returned by the factorization routine [tzrzf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result over C).

The matrix Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	<p>CHARACTER*1. Must be either 'L' or 'R'.</p> <p>If <i>side</i> = 'L', Q or Q^H is applied to C from the left.</p> <p>If <i>side</i> = 'R', Q or Q^H is applied to C from the right.</p>
<i>trans</i>	<p>CHARACTER*1. Must be either 'N' or 'C'.</p> <p>If <i>trans</i> = 'N', the routine multiplies C by Q.</p> <p>If <i>trans</i> = 'C', the routine multiplies C by Q^H.</p>
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>$0 \leq k \leq m$, if <i>side</i> = 'L';</p> <p>$0 \leq k \leq n$, if <i>side</i> = 'R'.</p>
<i>l</i>	<p>INTEGER.</p> <p>The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints:</p> <p>$0 \leq l \leq m$, if <i>side</i> = 'L';</p> <p>$0 \leq l \leq n$, if <i>side</i> = 'R'.</p>
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	<p>COMPLEX for cunmrz</p> <p>DOUBLE COMPLEX for zunmrz.</p> <p>Arrays: $a(lda,*)$, $tau(*)$, $c ldc,*)$, $work(lwork)$.</p> <p>On entry, the ith row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ctzrzf/ztzrzf in the last k rows of its array argument a.</p> <p>The second dimension of a must be at least $\max(1, m)$ if <i>side</i> = 'L', and at least $\max(1, n)$ if <i>side</i> = 'R'.</p> <p>$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ctzrzf/ztzrzf.</p> <p>The size of tau must be at least $\max(1, k)$.</p> <p>$c ldc,*)$ contains the m-by-n matrix C.</p> <p>The second dimension of c must be at least $\max(1, n)$</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if <i>side</i> = 'L'; $lwork \geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmrz` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k</i> , <i>m</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ (if *side* = 'L') or $lwork = m*blocksize$ (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormrz](#).

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

```
call sggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrf(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call ggqrf(a, b [,taua] [,taub] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the generalized QR factorization of an n -by- m matrix A and an n -by- p matrix B as $A = Q^*R$, $B = Q^*T^*Z$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} & & m \\ & m & \\ n - m & \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} & \end{matrix}, \quad \text{if } n \geq m$$

or

$$R = \begin{matrix} & n & m - n \\ n & \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} & \end{matrix}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{matrix} & p - n & n \\ n & \begin{pmatrix} 0 & T_{12} \end{pmatrix} & \end{matrix}, \quad \text{if } n \leq p,$$

$$T = \begin{matrix} & & p \\ n - p & - & p \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \end{matrix}, \quad \text{if } n > p,$$

where T_{12} or T_{21} is a p -by- p upper triangular matrix.

In particular, if B is square and nonsingular, the *GQR* factorization of A and B implicitly gives the *QR* factorization of $B^{-1}A$ as:

$$B^{-1}A = Z^T (T^{-1}R) \quad (\text{for real flavors}) \quad \text{or} \quad B^{-1}A = Z^{H*} (T^{-1}R) \quad (\text{for complex flavors}).$$

Input Parameters

n	INTEGER. The number of rows of the matrices A and B ($n \geq 0$).
m	INTEGER. The number of columns in A ($m \geq 0$).
p	INTEGER. The number of columns in B ($p \geq 0$).
$a, b, work$	REAL for <code>sggqrf</code> DOUBLE PRECISION for <code>dggqrf</code> COMPLEX for <code>cggqrf</code> DOUBLE COMPLEX for <code>zggqrf</code> . Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, m)$. $b(l db,*)$ contains the matrix B . The second dimension of b must be at least $\max(1, p)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
$lwork$	INTEGER. The size of the $work$ array; must be at least $\max(1, n, m, p)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

a, b	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array a contain the $\min(n,m)$ -by- m upper trapezoidal matrix R (R is upper triangular if $n \geq m$); the elements below the diagonal, with the array $taua$, represent the orthogonal/unitary matrix Q as a product of $\min(n,m)$ elementary reflectors ;
--------	---

if $n \leq p$, the upper triangle of the subarray $b(1:n, p-n+1:p)$ contains the n -by- n upper triangular matrix T ;
 if $n > p$, the elements on and above the $(n-p)$ th subdiagonal contain the n -by- p upper trapezoidal matrix T ; the remaining elements, with the array $taub$, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.

taua, taub REAL for sggqrf
 DOUBLE PRECISION for dggqrf
 COMPLEX for cggqrf
 DOUBLE COMPLEX for zggqrf.

Arrays, size at least $\max(1, \min(n, m))$ for *taua* and at least $\max(1, \min(n, p))$ for *taub*. The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q .
 The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z .

work(1) If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *ggqrf* interface are the following:

a Holds the matrix A of size (n,m) .
b Holds the matrix B of size (n,p) .
taua Holds the vector of length $\min(n,m)$.
taub Holds the vector of length $\min(n,p)$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1)H(2) \dots H(k), \text{ where } k = \min(n,m).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_a * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau_a * v * v^H \text{ for complex flavors,}$$

where τ_a is a real/complex scalar, and v is a real/complex vector with $v_j = 0$ for $1 \leq j \leq i - 1$, $v_i = 1$.

On exit, for $i + 1 \leq j \leq n$, v_j is stored in $a(i+1:n, i)$ and τ_a is stored in $taua(i)$

The matrix Z is represented as a product of elementary reflectors

$Z = H(1)H(2)\dots H(k)$, where $k = \min(n, p)$.

Each $H(i)$ has the form

$H(i) = I - \tau_b v v^T$ for real flavors, or

$H(i) = I - \tau_b v v^H$ for complex flavors,

where τ_b is a real/complex scalar, and v is a real/complex vector with $v_{p-k+1} = 1$, $v_j = 0$ for $p-k+1 \leq j \leq p-1$.

On exit, for $1 \leq j \leq p-k+i-1$, v_j is stored in $b(n-k+i, 1:p-k+i-1)$ and τ_b is stored in $taub(i)$.

For better performance, try using $lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3)$, where $nb1$ is the optimal blocksize for the QR factorization of an n -by- m matrix, $nb2$ is the optimal blocksize for the RQ factorization of an n -by- p matrix, and $nb3$ is the optimal blocksize for a call of `ormqr/unmqr`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggrqf

Computes the generalized RQ factorization of two matrices.

Syntax

```
call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call ggrqf(a, b [,taua] [,taub] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine forms the generalized RQ factorization of an m -by- n matrix A and an p -by- n matrix B as $A = R^*Q$, $B = Z^*T^*Q$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} n-m & m \\ m & \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } m \leq n,$$

or

$$R = \begin{matrix} m - n & & n \\ & & \\ & & \end{matrix} \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix}, \text{ if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} & & n \\ & n & \\ p - n & & \end{matrix} \begin{pmatrix} T_{11} \\ 0 \end{pmatrix}, \text{ if } p \geq n,$$

or

$$T = \begin{matrix} & p & n - p \\ p & (T_{11} & T_{12}) \end{matrix}, \text{ if } p < n,$$

where T_{11} is upper triangular.

In particular, if B is square and nonsingular, the GRQ factorization of A and B implicitly gives the RQ factorization of $A*B^{-1}$ as:

$$A*B^{-1} = (R*T^{-1}) * Z^T \text{ (for real flavors) or } A*B^{-1} = (R*T^{-1}) * Z^H \text{ (for complex flavors).}$$

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
p	INTEGER. The number of rows in B ($p \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
$a, b, work$	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $b(lb,*)$ contains the p -by- n matrix B . The second dimension of b must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, p)$.

lwork INTEGER. The size of the *work* array; must be at least $\max(1, n, m, p)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a, b Overwritten by the factorization data as follows:
 on exit, if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix *R*;
 if $m > n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix *R*;
 the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors.
 The elements on and above the diagonal of the array *b* contain the $\min(p, n)$ -by- n upper trapezoidal matrix *T* (*T* is upper triangular if $p \geq n$); the elements below the diagonal, with the array *taub*, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.

taua, taub REAL for *sggrqf*
 DOUBLE PRECISION for *dggrqf*
 COMPLEX for *cggrqf*
 DOUBLE COMPLEX for *zggrqf*.
 Arrays, size at least $\max(1, \min(m, n))$ for *taua* and at least $\max(1, \min(p, n))$ for *taub*.
 The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.
 The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

work(1) If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *ggrqf* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>A</i> of size (<i>p</i> , <i>n</i>).
<i>taua</i>	Holds the vector of length min(<i>m</i> , <i>n</i>).
<i>taub</i>	Holds the vector of length min(<i>p</i> , <i>n</i>).

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$$Q = H(1)H(2) \dots H(k), \text{ where } k = \min(m, n).$$

Each *H*(*i*) has the form

$$H(i) = I - \textit{taua} * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \textit{taua} * v * v^H \text{ for complex flavors,}$$

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v_{n-k+i} = 1, v_{n-k+i+1:n} = 0$.

On exit, $v_{1:n-k+i-1}$ is stored in $a(m-k+i, 1:n-k+i-1)$ and *taua* is stored in $\textit{taua}(i)$.

The matrix *Z* is represented as a product of elementary reflectors

$$Z = H(1)H(2) \dots H(k), \text{ where } k = \min(p, n).$$

Each *H*(*i*) has the form

$$H(i) = I - \textit{taub} * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \textit{taub} * v * v^H \text{ for complex flavors,}$$

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v_{1:i-1} = 0, v_i = 1$.

On exit, $v_{i+1:p}$ is stored in $b(i+1:p, i)$ and *taub* is stored in $\textit{taub}(i)$.

For better performance, try using

$$lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3),$$

where *nb1* is the optimal blocksize for the *RQ* factorization of an *m*-by-*n* matrix, *nb2* is the optimal blocksize for the *QR* factorization of an *p*-by-*n* matrix, and *nb3* is the optimal blocksize for a call of `?ormrq/?unmrq`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tpqrt

Computes a blocked QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for *Q*.

Syntax

```
call stpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call dtpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ctpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ztpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ttpqrt(a, b, t, nb[, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \left\{ \begin{array}{l} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{array} \right.$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-1)$ -by- n rectangular matrix $B1$ on top of an 1 -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \left\{ \begin{array}{l} \leftarrow (m-1) \times n \text{ rectangular} \\ \leftarrow 1 \times n \text{ upper trapezoidal} \end{array} \right.$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The elementary reflectors $H(i)$ are stored in the i th column below the diagonal in the $(n+m)$ -by- n input matrix C . The structure of vectors defining the elementary reflectors is illustrated by:

$$\begin{bmatrix} I \\ V \end{bmatrix} \left\{ \begin{array}{l} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{array} \right.$$

The elements of the unit matrix I are not stored. Thus, V contains all of the necessary information, and is returned in array b .

NOTE

Note that V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \left\{ \begin{array}{l} \leftarrow (m-1) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{array} \right.$$

The columns of V represent the vectors which define the $H(i)$ s.

The number of blocks is $k = \text{ceiling}(n/nb)$, where each block is of order nb except for the last block, which is of order $ib = n - (k-1)*nb$. For each of the k blocks, an upper triangular block reflector factor is computed: $T1, T2, \dots, Tk$. The nb -by- nb (ib -by- ib for the last block) T is are stored in the nb -by- n array t as

$$t = [T1T2 \dots Tk] .$$

Input Parameters

m	INTEGER. The total number of rows in the matrix B ($m \geq 0$).
n	INTEGER. The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
l	INTEGER. The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).
nb	INTEGER. The block size to use in the blocked QR factorization ($n \geq nb \geq 1$).
$a, b, work$	REAL for <code>stpqrt</code> DOUBLE PRECISION for <code>dtpqrt</code> COMPLEX for <code>ctpqrt</code> COMPLEX*16 for <code>ztpqrt</code> . Arrays: a size (lda, n) contains the n -by- n upper triangular matrix A . b size (ldb, n), the pentagonal m -by- n matrix B . The first $(m-1)$ rows contain the rectangular $B1$ matrix, and the next l rows contain the upper trapezoidal $B2$ matrix. $work$ size (nb, n) is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least nb .

Output Parameters

a	The elements on and above the diagonal of the array contain the upper triangular matrix R .
b	The pentagonal matrix V .
t	REAL for <code>stpqrt</code> DOUBLE PRECISION for <code>dtpqrt</code> COMPLEX for <code>ctpqrt</code> COMPLEX*16 for <code>ztpqrt</code> . Array, size (ldt, n). The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value.

?tpmqrt

Applies a real or complex orthogonal matrix obtained from a "triangular-pentagonal" complex block reflector to a general real or complex matrix, which consists of two blocks.

Syntax

```
call stpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call dtpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ctpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ztpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call tpmqrt( v, t, a, b, k, nb[, trans][, side][, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The columns of the pentagonal matrix V contain the elementary reflectors $H(1), H(2), \dots, H(k)$; V is composed of a rectangular block $V1$ and a trapezoidal block $V2$:

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$$

The size of the trapezoidal block $V2$ is determined by the parameter l , where $0 \leq l \leq k$. $V2$ is upper trapezoidal, consisting of the first l rows of a k -by- k upper triangular matrix.

If $l=k$, $V2$ is upper triangular;

If $l=0$, there is no trapezoidal block, so $V = V1$ is rectangular.

If *side* = 'L':

$$C = \begin{bmatrix} A \\ B \end{bmatrix}$$

where A is k -by- n , B is m -by- n and V is m -by- k .

If *side* = 'R':

$$C = [A \ B]$$

where A is m -by- k , B is m -by- n and V is n -by- k .

The real/complex orthogonal matrix Q is formed from V and T .

If $trans='N'$ and $side='L'$, c contains $Q * C$ on exit.

If $trans='T'$ and $side='L'$, C contains $Q^T * C$ on exit.

If $trans='C'$ and $side='L'$, C contains $Q^H * C$ on exit.

If $trans='N'$ and $side='R'$, C contains $C * Q$ on exit.

If $trans='T'$ and $side='R'$, C contains $C * Q^T$ on exit.

If $trans='C'$ and $side='R'$, C contains $C * Q^H$ on exit.

Input Parameters

<i>side</i>	CHARACTER*1 ='L': apply Q , Q^T , or Q^H from the left. ='R': apply Q , Q^T , or Q^H from the right.
<i>trans</i>	CHARACTER*1 ='N', no transpose, apply Q . ='T', transpose, apply Q^T . ='C', transpose, apply Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix B , ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix B , ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q , ($k \geq 0$).
<i>l</i>	INTEGER. The order of the trapezoidal part of V ($k \geq l \geq 0$).
<i>nb</i>	INTEGER. The block size used for the storage of t , $k \geq nb \geq 1$. This must be the same value of nb used to generate t in <code>tpqrt</code> .
<i>v</i>	REAL for <code>stpmqrt</code> DOUBLE PRECISION for <code>dtpmqrt</code> COMPLEX for <code>ctpmqrt</code> COMPLEX*16 for <code>ztpmqrt</code> . Size (ldv , k) The i th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>tpqrt</code> in array argument b .
<i>ldv</i>	INTEGER. The leading dimension of the array v . If $side = 'L'$, ldv must be at least $\max(1, m)$; If $side = 'R'$, ldv must be at least $\max(1, n)$.
<i>t</i>	REAL for <code>stpmqrt</code>

DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

Array, size (ldt , k).

The upper triangular factors of the block reflectors as returned by `tpqrt`, stored as an nb -by- k matrix.

ldt INTEGER. The leading dimension of the array *t*. *ldt* must be at least nb .

a REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

If *side* = 'L', size (lda , n).

If *side* = 'R', size (lda , k).

The k -by- n or m -by- k matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda* must be at least $\max(1,k)$.

If *side* = 'R', *lda* must be at least $\max(1,m)$.

b REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

Size (ldb , n).

The m -by- n matrix *B*.

ldb INTEGER. The leading dimension of the array *b*. *ldb* must be at least $\max(1,m)$.

work REAL for stpmqrt
DOUBLE PRECISION for dtpmqrt

COMPLEX for ctpmqrt

COMPLEX*16 for ztpmqrt.

Workspace array. If *side* = 'L' DIMENSION $n*nb$. If *side* = 'R' DIMENSION $m*nb$.

Output Parameters

a Overwritten by the corresponding block of the product Q^*C , C^*Q , Q^T*C , C^*Q^T , Q^H*C , or C^*Q^H .

b Overwritten by the corresponding block of the product Q^*C , C^*Q , Q^T^*C , C^*Q^T , Q^H^*C , or C^*Q^H .

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if *info* = -*i*, the *i*th argument had an illegal value.

Singular Value Decomposition - LAPACK Computational Routines

This section describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general *m*-by-*n* matrix *A*:

$$A = U\Sigma V^H.$$

In this decomposition, *U* and *V* are unitary (for complex *A*) or orthogonal (for real *A*); Σ is an *m*-by-*n* diagonal matrix with real diagonal elements σ_i :

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m, n)} \geq 0.$$

The diagonal elements σ_i are *singular values* of *A*. The first $\min(m, n)$ columns of the matrices *U* and *V* are, respectively, *left* and *right singular vectors* of *A*. The singular values and singular vectors satisfy

$$Av_i = \sigma_i u_i \text{ and } A^H u_i = \sigma_i v_i$$

where u_i and v_i are the *i*-th columns of *U* and *V*, respectively.

To find the SVD of a general matrix *A*, call the LAPACK routine ?gebrd or ?gbbbrd for reducing *A* to a bidiagonal matrix *B* by a unitary (orthogonal) transformation: $A = QBP^H$. Then call ?bdsqr, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of *A* is given by $A = U\Sigma V^H = (QU_1)\Sigma(V_1^H P^H)$.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists LAPACK routines (FORTRAN 77 interface) that perform singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce <i>A</i> to a bidiagonal matrix <i>B</i> : $A = QBP^H$ (full storage)	?gebrd	?gebrd
Reduce <i>A</i> to a bidiagonal matrix <i>B</i> : $A = QBP^H$ (band storage)	?gbbbrd	?gbbbrd
Generate the orthogonal (unitary) matrix <i>Q</i> or <i>P</i>	?orgbr	?ungbr
Apply the orthogonal (unitary) matrix <i>Q</i> or <i>P</i>	?ormbr	?unmbr
Form singular value decomposition of the bidiagonal matrix <i>B</i> : $B = U_1 \Sigma V_1^H$?bdsqr ?bdsdc	?bdsqr

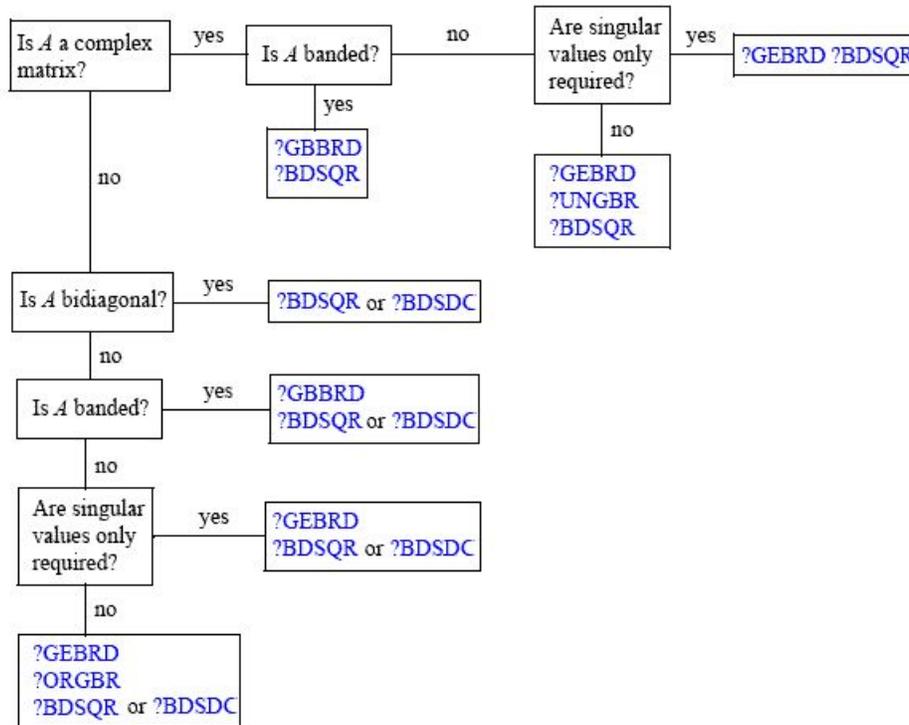
Decision Tree: Singular Value Decomposition

Figure "Decision Tree: Singular Value Decomposition" presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether A is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $\|Ax - b\|^2$. The effective rank k of the matrix A can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where Σ_k is the leading k -by- k submatrix of Σ , the matrix V_k consists of the first k columns of $V = PV_1$, and the vector c consists of the first k elements of $U^H b = U_1^H Q^H b$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```

call sgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call dgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call cgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call zgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call gebrd(a [, d] [, e] [, tauq] [, taup] [, info])
  
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a general m -by- n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

$$A = QB P^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P^H,$$

If $m \geq n$, the reduction is given by

where B_1 is an n -by- n upper diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; Q_1 consists of the first n columns of Q .

If $m < n$, the reduction is given by

$$A = Q^* B P^H = Q^* (B_1 0) P^H = Q_1^* B_1 P_1^H,$$

where B_1 is an m -by- m lower diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; P_1 consists of the first m columns of P .

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call [orgbr](#).
- to multiply a general matrix by Q or P , call [ormbr](#).

If the matrix A is complex,

- to compute Q and P explicitly, call [ungbr](#).
- to multiply a general matrix by Q or P , call [unmbr](#).

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for [sgebrd](#)
 DOUBLE PRECISION for [dgebrd](#)
 COMPLEX for [cgebrd](#)
 DOUBLE COMPLEX for [zgebrd](#).

Arrays:

$a(lda,*)$ contains the matrix A .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$lwork$ INTEGER.

The dimension of $work$; at least $\max(1, m, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	<p>If $m \geq n$, the diagonal and first super-diagonal of <i>a</i> are overwritten by the upper bidiagonal matrix <i>B</i>. The elements below the diagonal, with the array <i>tauq</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors, and the elements above the first superdiagonal, with the array <i>taup</i>, represent the orthogonal matrix <i>P</i> as a product of elementary reflectors.</p> <p>If $m < n$, the diagonal and first sub-diagonal of <i>a</i> are overwritten by the lower bidiagonal matrix <i>B</i>. The elements below the first subdiagonal, with the array <i>tauq</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors, and the elements above the diagonal, with the array <i>taup</i>, represent the orthogonal matrix <i>P</i> as a product of elementary reflectors.</p>
<i>d</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array, size at least $\max(1, \min(m, n))$.</p> <p>Contains the diagonal elements of <i>B</i>.</p>
<i>e</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of <i>B</i>.</p>
<i>tauq, taup</i>	<p>REAL for sgebrd</p> <p>DOUBLE PRECISION for dgebrd</p> <p>COMPLEX for cgebrd</p> <p>DOUBLE COMPLEX for zgebrd.</p> <p>Arrays, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrices <i>P</i> and <i>Q</i>.</p>
<i>work(1)</i>	<p>If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gebrd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>d</i>	Holds the vector of length $\min(m,n)$.
<i>e</i>	Holds the vector of length $\min(m,n)-1$.
<i>tauq</i>	Holds the vector of length $\min(m,n)$.
<i>taup</i>	Holds the vector of length $\min(m,n)$.

Application Notes

For better performance, try using $lwork = (m + n) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrices *Q*, *B*, and *P* satisfy $QB P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(4/3) * n^2 * (3 * m - n) \text{ for } m \geq n,$$

$$(4/3) * m^2 * (3 * n - m) \text{ for } m < n.$$

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling [geqrf](#) and then reduce the factor *R* to bidiagonal form. This requires approximately $2 * n^2 * (m + n)$ floating-point operations.

If *m* is much less than *n*, it can be more efficient to first form the *LQ* factorization of *A* by calling [gelqf](#) and then reduce the factor *L* to bidiagonal form. This requires approximately $2 * m^2 * (m + n)$ floating-point operations.

?gbbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

```
call sgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, info)
```

```
call dgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, info)
```

```
call cgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, rwork, info)
```

```
call zgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work, rwork, info)
```

```
call gbbbrd(ab [, c] [, d] [, e] [, q] [, pt] [, kl] [, m] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces an m -by- n band matrix A to upper bidiagonal matrix B : $A = Q^*B^*P^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H * C$.

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'N' or 'Q' or 'P' or 'B'. If <i>vect</i> = 'N', neither Q nor P^H is generated. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^H . If <i>vect</i> = 'B', the routine generates both Q and P^H .
<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>ncc</i>	INTEGER. The number of columns in C ($ncc \geq 0$).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
<i>ab, c, work</i>	REAL for sgbbrd DOUBLE PRECISION for dgbbrd COMPLEX for cgbbrd DOUBLE COMPLEX for zgbbrd. Arrays: <i>ab</i> (<i>ldab</i> ,*) contains the matrix A in band storage (see Matrix Storage Schemes). The second dimension of a must be at least $\max(1, n)$. <i>c</i> (<i>ldc</i> ,*) contains an m -by- ncc matrix C . If $ncc = 0$, the array c is not referenced. The second dimension of c must be at least $\max(1, ncc)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $2 * \max(m, n)$ for real flavors, or $\max(m, n)$ for complex flavors.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ($ldab \geq kl + ku + 1$).
<i>ldq</i>	INTEGER. The leading dimension of the output array q .

	$ldq \geq \max(1, m)$ if <i>vect</i> = 'Q' or 'B', $ldq \geq 1$ otherwise.
<i>ldpt</i>	INTEGER. The leading dimension of the output array <i>pt</i> . $ldpt \geq \max(1, n)$ if <i>vect</i> = 'P' or 'B', $ldpt \geq 1$ otherwise.
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$ if <i>ncc</i> > 0; $ldc \geq 1$ if <i>ncc</i> = 0.
<i>rwork</i>	REAL for <i>cgbbbrd</i> DOUBLE PRECISION for <i>zgbbbrd</i> . A workspace array, size at least $\max(m, n)$.

Output Parameters

<i>ab</i>	Overwritten by values generated during the reduction.
<i>d</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, size at least $\max(1, \min(m, n))$. Contains the diagonal elements of the matrix <i>B</i> .
<i>e</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of <i>B</i> .
<i>q, pt</i>	REAL for <i>sgebrd</i> DOUBLE PRECISION for <i>dgebrd</i> COMPLEX for <i>cgebrd</i> DOUBLE COMPLEX for <i>zgebrd</i> . Arrays: <i>q</i> (<i>ldq</i> , *) contains the output <i>m</i> -by- <i>m</i> matrix <i>Q</i> . The second dimension of <i>q</i> must be at least $\max(1, m)$. <i>p</i> (<i>ldpt</i> , *) contains the output <i>n</i> -by- <i>n</i> matrix <i>P</i> ^T . The second dimension of <i>pt</i> must be at least $\max(1, n)$.
<i>c</i>	Overwritten by the product $Q^{H*}C$. <i>c</i> is not referenced if <i>ncc</i> = 0.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gbbrrd` interface are the following:

<code>ab</code>	Holds the array A of size $(kl+ku+1,n)$.
<code>c</code>	Holds the matrix C of size (m,ncc) .
<code>d</code>	Holds the vector with the number of elements $\min(m,n)$.
<code>e</code>	Holds the vector with the number fo elements $\min(m,n)-1$.
<code>q</code>	Holds the matrix Q of size (m,m) .
<code>pt</code>	Holds the matrix PT of size (n,n) .
<code>m</code>	If omitted, assumed $m = n$.
<code>kl</code>	If omitted, assumed $kl = ku$.
<code>ku</code>	Restored as $ku = lda-kl-1$.
<code>vect</code>	Restored based on the presence of arguments <code>q</code> and <code>pt</code> as follows: <code>vect = 'B'</code> , if both <code>q</code> and <code>pt</code> are present, <code>vect = 'Q'</code> , if <code>q</code> is present and <code>pt</code> omitted, <code>vect = 'P'</code> , if <code>q</code> is omitted and <code>pt</code> present, <code>vect = 'N'</code> , if both <code>q</code> and <code>pt</code> are omitted.

Application Notes

The computed matrices Q , B , and P satisfy $Q^*B^*P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ε is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if `vect = 'N'` and `ncc = 0`,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if C is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either Q or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

```
call sorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call dorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call orgbr(a, tau [,vect] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines [gebrd/gebzd](#). Use this routine after a call to [sgebrd/dgebrd](#). All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m -by- m matrix Q :

```
call ?orgbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
call ?orgbr('Q', m, n, n, a ... )
```

To compute the whole n -by- n matrix P^T :

```
call ?orgbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
call ?orgbr('P', m, n, m, a ... )
```

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'Q' or 'P'. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^T .
<i>m, n</i>	INTEGER. The number of rows (m) and columns (n) in the matrix Q or P^T to be returned ($m \geq 0, n \geq 0$). If <i>vect</i> = 'Q', $m \geq n \geq \min(m, k)$. If <i>vect</i> = 'P', $n \geq m \geq \min(n, k)$.
<i>k</i>	If <i>vect</i> = 'Q', the number of columns in the original m -by- k matrix reduced by gebrd . If <i>vect</i> = 'P', the number of rows in the original k -by- n matrix reduced by gebrd .
<i>a</i>	REAL for <code>sorgbr</code> DOUBLE PRECISION for <code>dorgbr</code> The vectors which define the elementary reflectors, as returned by gebrd .
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for <code>sorgbr</code> DOUBLE PRECISION for <code>dorgbr</code>

Array, size $\min(m, k)$ if $vect = 'Q'$, $\min(n, k)$ if $vect = 'P'$.
 Scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q and P^T as returned by [gebrd](#) in the array $tauq$ or $taup$.

work REAL for `sorgbr`

DOUBLE PRECISION for `dorgbr`

Workspace array, size $\max(1, lwork)$.

lwork INTEGER. Dimension of the array *work*. See *Application Notes* for the suggested value of *lwork*.

If $lwork = -1$ then the routine performs a workspace query and calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

Output Parameters

a Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, m , and n .

work(1) If $info = 0$, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orgbr` interface are the following:

a Holds the matrix A of size (m, n) .

tau Holds the vector of length $\min(m, k)$ where

$k = m$, if $vect = 'P'$,

$k = n$, if $vect = 'Q'$.

vect Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using $lwork = \min(m, n) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of *Q*:

$$(4/3) * n * (3m^2 - 3m * n + n^2) \text{ if } m > n;$$

$$(4/3) * m^3 \text{ if } m \leq n.$$

To form the *n* leading columns of *Q* when $m > n$:

$$(2/3) * n^2 * (3m - n^2) \text{ if } m > n.$$

To form the whole of P^T :

$$(4/3) * n^3 \text{ if } m \geq n;$$

$$(4/3) * m * (3n^2 - 3m * n + m^2) \text{ if } m < n.$$

To form the *m* leading columns of P^T when $m < n$:

$$(2/3) * n^2 * (3m - n^2) \text{ if } m > n.$$

The complex counterpart of this routine is [ungbr](#).

?ormbr

*Multiplies an arbitrary real matrix by the real orthogonal matrix *Q* or P^T determined by ?gebrd.*

Syntax

```
call sormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call ormbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

Given an arbitrary real matrix *C*, this routine forms one of the matrix products $Q * C$, $Q^T * C$, $C * Q$, $C * Q^T$, $P * C$, $P^T * C$, $C * P$, $C * P^T$, where *Q* and *P* are orthogonal matrices computed by a call to [gebrd](#). The routine overwrites the product on *C*.

Input Parameters

In the descriptions below, *r* denotes the order of *Q* or P^T :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>vect</i>	CHARACTER*1. Must be 'Q' or 'P'. If <i>vect</i> = 'Q', then <i>Q</i> or Q^T is applied to <i>C</i> . If <i>vect</i> = 'P', then <i>P</i> or P^T is applied to <i>C</i> .
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'.

If *side* = 'L', multipliers are applied to *C* from the left.
 If *side* = 'R', they are applied to *C* from the right.

trans CHARACTER*1. Must be 'N' or 'T'.
 If *trans* = 'N', then *Q* or *P* is applied to *C*.
 If *trans* = 'T', then Q^T or P^T is applied to *C*.

m INTEGER. The number of rows in *C*.

n INTEGER. The number of columns in *C*.

k INTEGER. One of the dimensions of *A* in ?gebrd:
 If *vect* = 'Q', the number of columns in *A*;
 If *vect* = 'P', the number of rows in *A*.
 Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

a, c, work REAL for sormbr
 DOUBLE PRECISION for dormbr.
 Arrays:
a(lda,)* is the array *a* as returned by ?gebrd.
 Its second dimension must be at least $\max(1, \min(r, k))$ for *vect* = 'Q', or $\max(1, r)$ for *vect* = 'P'.
c ldc,)* holds the matrix *C*.
 Its second dimension must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*. Constraints:
lda $\geq \max(1, r)$ if *vect* = 'Q';
lda $\geq \max(1, \min(r, k))$ if *vect* = 'P'.

ldc INTEGER. The leading dimension of *c*; *ldc* $\geq \max(1, m)$.

tau REAL for sormbr
 DOUBLE PRECISION for dormbr.
 Array, size at least $\max(1, \min(r, k))$.
 For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P',
 the array *taup* as returned by ?gebrd.

lwork INTEGER. The size of the *work* array. Constraints:
lwork $\geq \max(1, n)$ if *side* = 'L';
lwork $\geq \max(1, m)$ if *side* = 'R'.
 If *lwork* = -1, then a workspace query is assumed; the routine only
 calculates the optimal size of the *work* array, returns this value as the first
 entry of the *work* array, and no error message related to *lwork* is issued by
[xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^{T*}C$, C^*Q , C^*Q^T , P^*C , $P^{T*}C$, C^*P , or C^*P^T , as specified by <i>vect</i> , <i>side</i> , and <i>trans</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormbr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size $(r, \min(nq, k))$ where $r = nq$, if <i>vect</i> = 'Q', $r = \min(nq, k)$, if <i>vect</i> = 'P', $nq = m$, if <i>side</i> = 'L', $nq = n$, if <i>side</i> = 'R', $k = m$, if <i>vect</i> = 'P', $k = n$, if <i>vect</i> = 'Q'.
<i>tau</i>	Holds the vector of length $\min(nq, k)$.
<i>c</i>	Holds the matrix <i>C</i> of size (m, n) .
<i>vect</i>	Must be 'Q' or 'P'. The default value is 'Q'.
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using

lwork = *n***blocksize* for *side* = 'L', or

lwork = *m***blocksize* for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

$2 * n * k (2 * m - k)$ if *side* = 'L' and $m \geq k$;

$2 * m * k (2 * n - k)$ if *side* = 'R' and $n \geq k$;

$2 * m^2 * n$ if *side* = 'L' and $m < k$;

$2 * n^2 * m$ if *side* = 'R' and $n < k$.

The complex counterpart of this routine is [unmbr](#).

?ungbr

Generates the complex unitary matrix *Q* or P^H determined by ?gebrd.

Syntax

```
call cungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

```
call zungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

```
call ungbr(a, tau [,vect] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine generates the whole or part of the unitary matrices *Q* and P^H formed by the routines [gebrd/](#)[gebrd](#). Use this routine after a call to [cgebrd/zgebrd](#). All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole *m*-by-*m* matrix *Q*, use:

```
call ?ungbr('Q', m, m, n, a ... )
```

(note that the array *a* must have at least *m* columns).

To form the *n* leading columns of *Q* if $m > n$, use:

```
call ?ungbr('Q', m, n, n, a ... )
```

To compute the whole *n*-by-*n* matrix P^H , use:

```
call ?ungbr('P', n, n, m, a ... )
```

(note that the array *a* must have at least *n* rows).

To form the *m* leading rows of P^H if $m < n$, use:

```
call ?ungbr('P', m, n, m, a ... )
```

Input Parameters

<i>vect</i>	<p>CHARACTER*1. Must be 'Q' or 'P'.</p> <p>If <i>vect</i> = 'Q', the routine generates the matrix Q.</p> <p>If <i>vect</i> = 'P', the routine generates the matrix P^H.</p>
<i>m</i>	INTEGER. The number of required rows of Q or P^H .
<i>n</i>	INTEGER. The number of required columns of Q or P^H .
<i>k</i>	<p>INTEGER. One of the dimensions of A in ?gebrd:</p> <p>If <i>vect</i> = 'Q', the number of columns in A;</p> <p>If <i>vect</i> = 'P', the number of rows in A.</p> <p>Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.</p> <p>For <i>vect</i> = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$.</p> <p>For <i>vect</i> = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.</p>
<i>a, work</i>	<p>COMPLEX for <i>cungbr</i></p> <p>DOUBLE COMPLEX for <i>zungbr</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> is the array <i>a</i> as returned by ?gebrd.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>tau</i>	<p>COMPLEX for <i>cungbr</i></p> <p>DOUBLE COMPLEX for <i>zungbr</i>.</p> <p>For <i>vect</i> = 'Q', the array <i>tauq</i> as returned by ?gebrd. For <i>vect</i> = 'P', the array <i>taup</i> as returned by ?gebrd.</p> <p>The dimension of <i>tau</i> must be at least $\max(1, \min(m, k))$ for <i>vect</i> = 'Q', or $\max(1, \min(m, k))$ for <i>vect</i> = 'P'.</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array.</p> <p>Constraint: $lwork < \max(1, \min(m, n))$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by <i>vect</i> , <i>m</i> , and <i>n</i> .
----------	--

<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ungbr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length $\min(m,k)$ where $k = m$, if <i>vect</i> = 'P', $k = n$, if <i>vect</i> = 'Q'.
<i>vect</i>	Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using $lwork = \min(m,n) * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix *Q*:

$$(16/3)n(3m^2 - 3m*n + n^2) \text{ if } m > n;$$

$$(16/3)m^3 \text{ if } m \leq n.$$

To form the *n* leading columns of *Q* when $m > n$:

$$(8/3)n^2(3m - n^2).$$

To compute the whole matrix P^H :

$$(16/3)n^3 \text{ if } m \geq n;$$

$$(16/3)m(3n^2 - 3m*n + m^2) \text{ if } m < n.$$

To form the *m* leading columns of P^H when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is [orgbr](#).

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

```
call cunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call unmbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

Given an arbitrary complex matrix C , this routine forms one of the matrix products Q^*C , Q^H*C , $C*Q$, $C*Q^H$, P^*C , P^H*C , $C*P$, or $C*P^H$, where Q and P are unitary matrices computed by a call to [gebrd/gebrd](#). The routine overwrites the product on C .

Input Parameters

In the descriptions below, r denotes the order of Q or P^H :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>vect</i>	CHARACTER*1. Must be 'Q' or 'P'. If <i>vect</i> = 'Q', then Q or Q^H is applied to C . If <i>vect</i> = 'P', then P or P^H is applied to C .
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', multipliers are applied to C from the left. If <i>side</i> = 'R', they are applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q or P is applied to C . If <i>trans</i> = 'C', then Q^H or P^H is applied to C .
<i>m</i>	INTEGER. The number of rows in C .
<i>n</i>	INTEGER. The number of columns in C .
<i>k</i>	INTEGER. One of the dimensions of A in ?gebrd: If <i>vect</i> = 'Q', the number of columns in A ; If <i>vect</i> = 'P', the number of rows in A . Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
<i>a, c, work</i>	COMPLEX for cunmbr

DOUBLE COMPLEX for `zunmbr`.

Arrays:

$a(lda,*)$ is the array a as returned by `?gebrd`.

Its second dimension must be at least $\max(1, \min(r,k))$ for $vect = 'Q'$, or $\max(1, r)$ for $vect = 'P'$.

$c ldc,*)$ holds the matrix C .

Its second dimension must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a . Constraints:

$lda \geq \max(1, r)$ if $vect = 'Q'$;

$lda \geq \max(1, \min(r,k))$ if $vect = 'P'$.

ldc

INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

tau

COMPLEX for `cunmbr`

DOUBLE COMPLEX for `zunmbr`.

Array, size at least $\max(1, \min(r, k))$.

For $vect = 'Q'$, the array $tauq$ as returned by `?gebrd`. For $vect = 'P'$, the array $taup$ as returned by `?gebrd`.

$lwork$

INTEGER. The size of the $work$ array.

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

$lwork \geq 1$ if $n=0$ or $m=0$.

For optimum performance $lwork \geq \max(1, n*nb)$ if $side = 'L'$, and $lwork \geq \max(1, m*nb)$ if $side = 'R'$, where nb is the optimal blocksize. ($nb = 0$ if $m = 0$ or $n = 0$.)

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c

Overwritten by the product $Q*C$, Q^H*C , $C*Q$, $C*Q^H$, $P*C$, P^H*C , $C*P$, or $C*P^H$, as specified by $vect$, $side$, and $trans$.

$work(1)$

If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmbr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size $(r, \min(nq, k))$ where $r = nq$, if <i>vect</i> = 'Q', $r = \min(nq, k)$, if <i>vect</i> = 'P', $nq = m$, if <i>side</i> = 'L', $nq = n$, if <i>side</i> = 'R', $k = m$, if <i>vect</i> = 'P', $k = n$, if <i>vect</i> = 'Q'.
<i>tau</i>	Holds the vector of length $\min(nq, k)$.
<i>c</i>	Holds the matrix <i>C</i> of size (m, n) .
<i>vect</i>	Must be 'Q' or 'P'. The default value is 'Q'.
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, use

$lwork = n * blocksize$ for *side* = 'L', or

$lwork = m * blocksize$ for *side* = 'R',

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

$8 * n * k (2 * m - k)$ if *side* = 'L' and $m \geq k$;

$8 * m * k (2 * n - k)$ if *side* = 'R' and $n \geq k$;

$8 * m^2 * n$ if *side* = 'L' and $m < k$;

$8 * n^2 * m$ if *side* = 'R' and $n < k$.

The real counterpart of this routine is [ormbr](#).

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

```
call sbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call dbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call cbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
call zbdsqr(uplo, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
call rbdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
call bdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular values and, optionally, the right and/or left singular vectors from the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q^* S P^H$ where S is the diagonal matrix of singular values, Q is an orthogonal matrix of left singular vectors, and P is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U^* Q$ instead of Q , and, if right singular vectors are requested, this subroutine returns $P^H * VT$ instead of P^H , for given real/complex input matrices U and VT . When U and VT are the orthogonal/unitary matrices that reduce a general matrix A to bidiagonal form: $A = U^* B^* VT$, as computed by [?gebrd](#), then

$$A = (U^* Q) * S * (P^H * VT)$$

is the SVD of A . Optionally, the subroutine may also compute $Q^H * C$ for a given real/complex input matrix C . See also [lasq1](#), [lasq2](#), [lasq3](#), [lasq4](#), [lasq5](#), [lasq6](#) used by this routine.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', B is an upper bidiagonal matrix. If <i>uplo</i> = 'L', B is a lower bidiagonal matrix.
<i>n</i>	INTEGER. The order of the matrix B ($n \geq 0$).
<i>ncvt</i>	INTEGER. The number of columns of the matrix VT , that is, the number of right singular vectors ($ncvt \geq 0$). Set <i>ncvt</i> = 0 if no right singular vectors are required.
<i>nru</i>	INTEGER. The number of rows in U , that is, the number of left singular vectors ($nru \geq 0$). Set <i>nru</i> = 0 if no left singular vectors are required.

<i>ncc</i>	<p>INTEGER. The number of columns in the matrix <i>C</i> used for computing the product $Q^H * C$ ($ncc \geq 0$). Set $ncc = 0$ if no matrix <i>C</i> is supplied.</p>
<i>d, e</i>	<p>REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the diagonal elements of <i>B</i>. The size of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the $(n-1)$ off-diagonal elements of <i>B</i>. The size of <i>e</i> must be at least $\max(1, n - 1)$.</p>
<i>work</i>	<p>REAL for sbdsqr DOUBLE PRECISION for dbdsqr.</p> <p><i>work</i>(*) is a workspace array. The size of <i>work</i> must be at least $\max(1, 4 * n)$.</p>
<i>rwork</i>	<p>REAL for cbdsqr DOUBLE PRECISION for zbdsqr.</p> <p><i>rwork</i>(*) is a workspace array. The size of <i>rwork</i> must be at least $\max(1, 4 * n)$.</p>
<i>vt, u, c</i>	<p>REAL for sbdsqr DOUBLE PRECISION for dbdsqr COMPLEX for cbdsqr DOUBLE COMPLEX for zbdsqr.</p> <p>Arrays:</p> <p><i>vt</i>(<i>ldvt</i>,*) contains an n-by-<i>ncvt</i> matrix <i>VT</i>. The second dimension of <i>vt</i> must be at least $\max(1, ncvt)$. <i>vt</i> is not referenced if $ncvt = 0$.</p> <p><i>u</i>(<i>ldu</i>,*) contains an <i>nru</i> by n matrix <i>U</i>. The second dimension of <i>u</i> must be at least $\max(1, n)$. <i>u</i> is not referenced if $nru = 0$.</p> <p><i>c</i>(<i>ldc</i>,*) contains the n-by-<i>ncc</i> matrix <i>C</i> for computing the product $Q^H * C$. The second dimension of <i>c</i> must be at least $\max(1, ncc)$. The array is not referenced if $ncc = 0$.</p>
<i>ldvt</i>	<p>INTEGER. The leading dimension of <i>vt</i>. Constraints: $ldvt \geq \max(1, n)$ if $ncvt > 0$; $ldvt \geq 1$ if $ncvt = 0$.</p>
<i>ldu</i>	<p>INTEGER. The leading dimension of <i>u</i>. Constraint:</p>

$ldu \geq \max(1, nru)$.

ldc INTEGER. The leading dimension of *c*. Constraints:
 $ldc \geq \max(1, n)$ if $ncc > 0$; $ldc \geq 1$ otherwise.

Output Parameters

d On exit, if $info = 0$, overwritten by the singular values in decreasing order (see *info*).

e On exit, if $info = 0$, *e* is destroyed. See also *info* below.

c Overwritten by the product $Q^H * C$.

vt On exit, this array is overwritten by $P^H * VT$. Not referenced if $ncvt = 0$.

u On exit, this array is overwritten by $U * Q$. Not referenced if $nru = 0$.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the *i*-th parameter had an illegal value.

If $info > 0$,

If $ncvt = nru = ncc = 0$,

- $info = 1$, a split was marked by a positive value in *e*
- $info = 2$, the current block of *z* not diagonalized after $100 * n$ iterations (in the inner `while` loop)
- $info = 3$, termination criterion of the outer `while` loop is not met (the program created more than *n* unreduced blocks).

In all other cases when $ncvt$, nru , or $ncc > 0$, the algorithm did not converge; *d* and *e* contain the elements of a bidiagonal matrix that is orthogonally similar to the input matrix *B*; if $info = i$, *i* elements of *e* have not converged to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `bdsqr` interface are the following:

d Holds the vector of length (*n*).

e Holds the vector of length (*n*).

vt Holds the matrix *VT* of size (*n*, *ncvt*).

u Holds the matrix *U* of size (*nru*, *n*).

c Holds the matrix *C* of size (*n*, *ncc*).

uplo Must be 'U' or 'L'. The default value is 'U'.

<i>ncvt</i>	If argument <i>vt</i> is present, then <i>ncvt</i> is equal to the number of columns in matrix <i>VT</i> ; otherwise, <i>ncvt</i> is set to zero.
<i>nru</i>	If argument <i>u</i> is present, then <i>nru</i> is equal to the number of rows in matrix <i>U</i> ; otherwise, <i>nru</i> is set to zero.
<i>ncc</i>	If argument <i>c</i> is present, then <i>ncc</i> is equal to the number of columns in matrix <i>C</i> ; otherwise, <i>ncc</i> is set to zero.

Note that two variants of Fortran 95 interface for `bdsqr` routine are needed because of an ambiguous choice between real and complex cases appear when *vt*, *u*, and *c* are omitted. Thus, the name `rbdsqr` is used in real cases (single or double precision), and the name `bdsqr` is used in complex cases (single or double precision).

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of B , and s_i is the corresponding computed value, then

$$|s_i - \sigma_i| \leq p(m, n) * \varepsilon * \sigma_i$$

where $p(m, n)$ is a modestly increasing function of m and n , and ε is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B , and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n) * \varepsilon / \min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|).$$

Here $\min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2 * nru$ additional operations ($12n^2 * nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2 * ncvt$ operations ($12n^2 * ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

```
call sbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call dbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call bdsdc(d, e [,u] [,vt] [,q] [,iq] [,uplo] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B : $B = U \Sigma V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and V are orthogonal matrices of left and right singular vectors, respectively. `?bdsdc` can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This routine uses `?lasd0`, `?lasd1`, `?lasd2`, `?lasd3`, `?lasd4`, `?lasd5`, `?lasd6`, `?lasd7`, `?lasd8`, `?lasd9`, `?lasda`, `?lasdq`, `?lasdt`.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', B is an upper bidiagonal matrix. If <i>uplo</i> = 'L', B is a lower bidiagonal matrix.
<i>compq</i>	CHARACTER*1. Must be 'N', 'P', or 'I'. If <i>compq</i> = 'N', compute singular values only. If <i>compq</i> = 'P', compute singular values and compute singular vectors in compact form. If <i>compq</i> = 'I', compute singular values and singular vectors.
<i>n</i>	INTEGER. The order of the matrix B ($n \geq 0$).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for <code>sbdsc</code> DOUBLE PRECISION for <code>dbdsc</code> . Arrays: <i>d</i> (*) contains the n diagonal elements of the bidiagonal matrix B . The size of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of the bidiagonal matrix B . The size of <i>e</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least: $\max(1, 4*n)$, if <i>compq</i> = 'N' or <i>compq</i> = 'P'; $\max(1, 3*n^2+4*n)$, if <i>compq</i> = 'I'.
<i>ldu</i>	INTEGER. The leading dimension of the output array <i>u</i> ; $ldu \geq 1$. If singular vectors are desired, then $ldu \geq \max(1, n)$.
<i>ldvt</i>	INTEGER. The leading dimension of the output array <i>vt</i> ; $ldvt \geq 1$. If singular vectors are desired, then $ldvt \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, dimension at least $\max(1, 8*n)$.

Output Parameters

<i>d</i>	If <i>info</i> = 0, overwritten by the singular values of B .
----------	---

<i>e</i>	On exit, <i>e</i> is overwritten.
<i>u, vt, q</i>	<p>REAL for <code>sbdsdc</code></p> <p>DOUBLE PRECISION for <code>dbdsdc</code>.</p> <p>Arrays: <i>u</i>(<i>ldu</i>, *), <i>vt</i>(<i>ldvt</i>, *), <i>q</i>(*).</p> <p>If <code>compq = 'I'</code>, then on exit <i>u</i> contains the left singular vectors of the bidiagonal matrix <i>B</i>, unless <code>info ≠ 0</code> (<i>seeinfo</i>). For other values of <code>compq</code>, <i>u</i> is not referenced.</p> <p>The second dimension of <i>u</i> must be at least <code>max(1,n)</code>.</p> <p>if <code>compq = 'I'</code>, then on exit <i>vt</i>^T contains the right singular vectors of the bidiagonal matrix <i>B</i>, unless <code>info ≠ 0</code> (<i>seeinfo</i>). For other values of <code>compq</code>, <i>vt</i> is not referenced. The second dimension of <i>vt</i> must be at least <code>max(1,n)</code>.</p> <p>If <code>compq = 'P'</code>, then on exit, if <code>info = 0</code>, <i>q</i> and <i>iq</i> contain the left and right singular vectors in a compact form. Specifically, <i>q</i> contains all the REAL (for <code>sbdsdc</code>) or DOUBLE PRECISION (for <code>dbdsdc</code>) data for singular vectors. For other values of <code>compq</code>, <i>q</i> is not referenced.</p>
<i>iq</i>	<p>INTEGER.</p> <p>Array: <i>iq</i>(*).</p> <p>If <code>compq = 'P'</code>, then on exit, if <code>info = 0</code>, <i>q</i> and <i>iq</i> contain the left and right singular vectors in a compact form. Specifically, <i>iq</i> contains all the INTEGER data for singular vectors. For other values of <code>compq</code>, <i>iq</i> is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <code>info = 0</code>, the execution is successful.</p> <p>If <code>info = -i</code>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <code>info = i</code>, the algorithm failed to compute a singular value. The update process of divide and conquer failed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `bdsdc` interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length <i>n</i> .
<i>u</i>	Holds the matrix <i>U</i> of size (<i>n,n</i>).
<i>vt</i>	Holds the matrix <i>VT</i> of size (<i>n,n</i>).
<i>q</i>	<p>Holds the vector of length (<i>ldq</i>), where</p> <p>$ldq \geq n * (11 + 2 * smlsiz + 8 * int(log_2(n / (smlsiz + 1))))$ and <i>smlsiz</i> is returned by <i>ilaenv</i> and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).</p>

compq

Restored based on the presence of arguments u , vt , q , and iq as follows:

$compq = 'N'$, if none of u , vt , q , and iq are present,

$compq = 'I'$, if both u and vt are present. Arguments u and vt must either be both present or both omitted,

$compq = 'P'$, if both q and iq are present. Arguments q and iq must either be both present or both omitted.

Note that there will be an error condition if all of u , vt , q , and iq arguments are present simultaneously.

See Also

[?lasd0](#)

[?lasd1](#)

[?lasd2](#)

[?lasd3](#)

[?lasd4](#)

[?lasd5](#)

[?lasd6](#)

[?lasd7](#)

[?lasd8](#)

[?lasd9](#)

[?lasda](#)

[?lasdq](#)

[?lasdt](#)

Symmetric Eigenvalue Problems

Symmetric eigenvalue problems are posed as follows: given an n -by- n real symmetric or complex Hermitian matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (or, equivalently, } z^H A = \lambda z^H \text{)}.$$

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A , and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

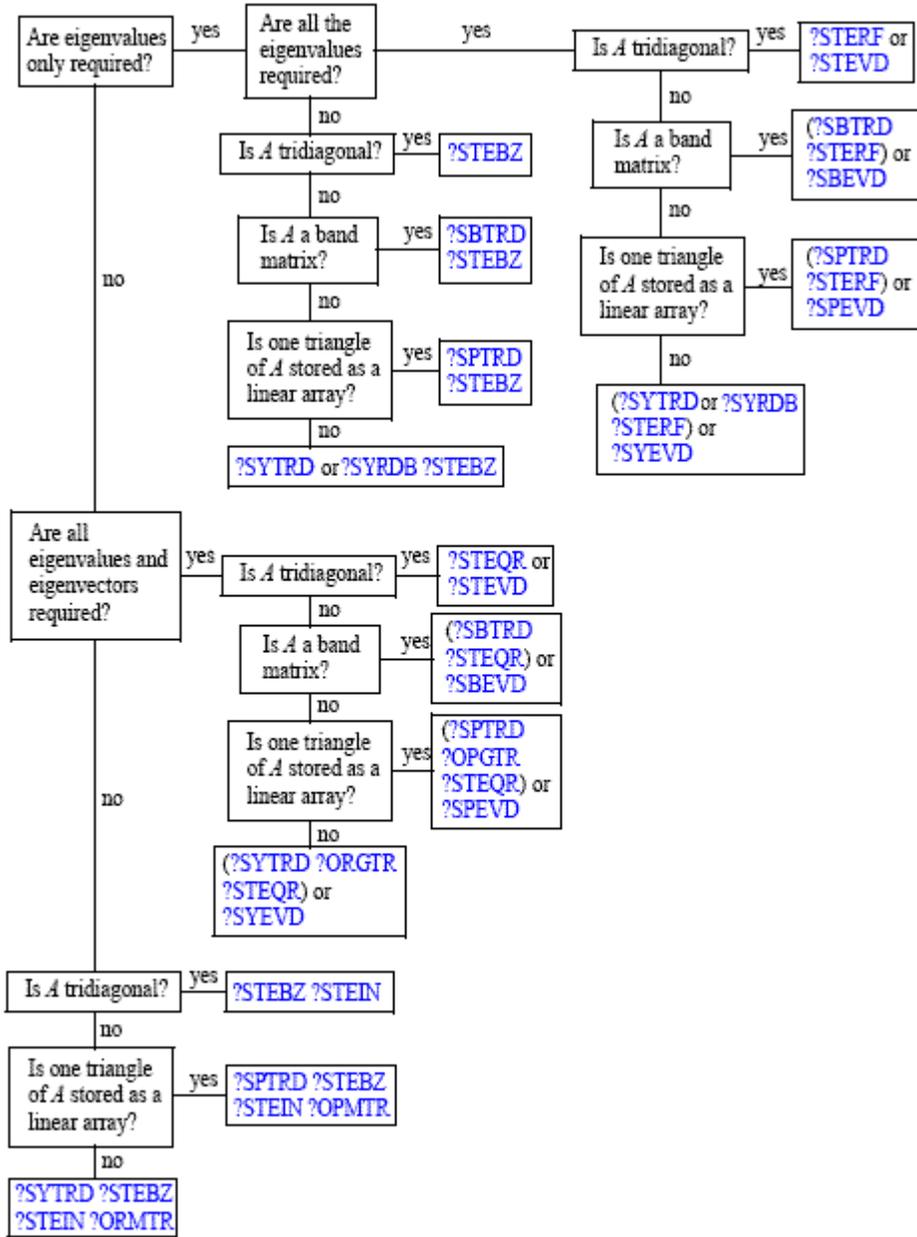
To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines (for FORTRAN 77 interface) are listed in [Table "Computational Routines for Solving Symmetric Eigenvalue Problems"](#). The corresponding routine names in the Fortran 95 interface are without the first symbol.

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.

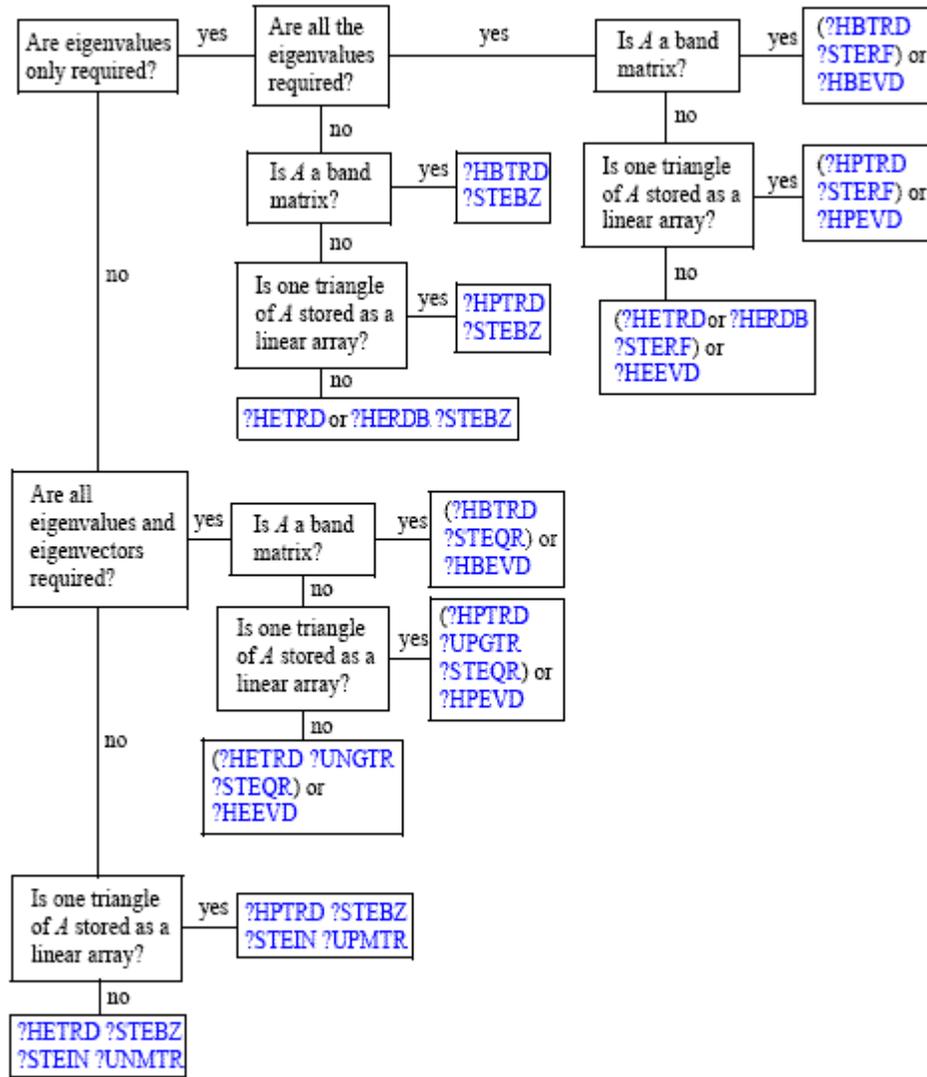
These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

The decision tree in [Figure "Decision Tree: Real Symmetric Eigenvalue Problems"](#) will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. [Figure "Decision Tree: Complex Hermitian Eigenvalue Problems"](#) presents a similar decision tree for complex Hermitian matrices.

: Decision Tree: Real Symmetric Eigenvalue Problems



: Decision Tree: Complex Hermitian Eigenvalue Problems



Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	sytrd syrdb	hetrd herdb
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	sptrd	hptrd
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	sbtrd	hbtrd
Generate matrix Q (full storage)	orgtr	ungtr
Generate matrix Q (packed storage)	opgtr	upgtr
Apply matrix Q (full storage)	ormtr	unmtr
Apply matrix Q (packed storage)	opmtr	upmtr

Operation	Real symmetric matrices	Complex Hermitian matrices
Find all eigenvalues of a tridiagonal matrix T	sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T	steqr stedc	steqr stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix T .	pteqr	pteqr
Find selected eigenvalues of a tridiagonal matrix T	stebz stegr	stegr
Find selected eigenvectors of a tridiagonal matrix T	stein stegr	stein stegr
Find selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T	stemr	stemr
Compute the reciprocal condition numbers for the eigenvectors	disna	disna

?sytrd

Reduces a real symmetric matrix to tridiagonal form.

Syntax

```
call ssytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call sytrd(a, tau [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^T T Q$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

`uplo` CHARACTER*1. Must be 'U' or 'L'.
 If `uplo` = 'U', `a` stores the upper triangular part of A .
 If `uplo` = 'L', `a` stores the lower triangular part of A .

`n` INTEGER. The order of the matrix A ($n \geq 0$).

`a, work` REAL for `ssytrd`
 DOUBLE PRECISION for `dsytrd`.

$a(lda,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by $uplo$. If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq n$).
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a On exit,
if $uplo = 'U'$, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array tau , represent the orthogonal matrix Q as a product of elementary reflectors;
if $uplo = 'L'$, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array tau , represent the orthogonal matrix Q as a product of elementary reflectors.

d, e, tau REAL for `ssytrd`
DOUBLE PRECISION for `dsytrd`.

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The size of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The size of e must be at least $\max(1, n-1)$.

$tau(*)$ stores $(n-1)$ scalars that define elementary reflectors in decomposition of the orthogonal matrix Q in a product of $n-1$ elementary reflectors. $tau(n)$ is used as workspace.

The size of tau must be at least $\max(1, n)$.

$work(1)$ If $info=0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.
If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sytrd` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

<code>orgtr</code>	to form the computed matrix Q explicitly
<code>ormtr</code>	to multiply a real matrix by Q .

The complex counterpart of this routine is [?hetrd](#).

?syldb

Reduces a real symmetric matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

```
call ssyldb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call dsyldb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- `mk1.fi`

Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^*T^*Q^T$ and optionally multiplies matrix Z by Q , or simply forms Q .

This routine reduces a full symmetric matrix A to the banded symmetric matrix B , and then to the tridiagonal symmetric matrix T with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [Bischof00]). `?syrd` is functionally close to `?sytrd` routine but the tridiagonal form may differ from those obtained by `?sytrd`. Unlike `?sytrd`, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only A is reduced to T . If <i>jobz</i> = 'V', then A is reduced to T and A contains Q on exit. If <i>jobz</i> = 'U', then A is reduced to T and Z contains Z^*Q on exit.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$, $kd \leq n-1$).
<i>a, z, work</i>	REAL for <code>ssyrd</code> . DOUBLE PRECISION for <code>dsyrd</code> . $a(lda,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by <i>uplo</i> . The second dimension of a must be at least $\max(1, n)$. $z(ldz,*)$, the second dimension of z must be at least $\max(1, n)$. If <i>jobz</i> = 'U', then the matrix z is multiplied by Q . If <i>jobz</i> = 'N' or 'V', then z is not referenced. <i>work(lwork)</i> is a workspace array.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ldz</i>	INTEGER. The leading dimension of z ; at least $\max(1, n)$. Not referenced if <i>jobz</i> = 'N'
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq (2kd+1)n+kd$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	<p>If <i>jobz</i> = 'V', then overwritten by <i>Q</i> matrix.</p> <p>If <i>jobz</i> = 'N' or 'U', then overwritten by the banded matrix <i>B</i> and details of the orthogonal matrix Q_B to reduce <i>A</i> to <i>B</i> as specified by <i>uplo</i>.</p>
<i>z</i>	<p>On exit,</p> <p>if <i>jobz</i> = 'U', then the matrix <i>z</i> is overwritten by $Z*Q$.</p> <p>If <i>jobz</i> = 'N' or 'V', then <i>z</i> is not referenced.</p>
<i>d, e, tau</i>	<p>DOUBLE PRECISION.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the diagonal elements of the matrix <i>T</i>. The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the off-diagonal elements of <i>T</i>. The dimension of <i>e</i> must be at least $\max(1, n-1)$.</p> <p><i>tau</i>(*) stores further details of the orthogonal matrix <i>Q</i>. The dimension of <i>tau</i> must be at least $\max(1, n-kd-1)$.</p>
<i>work</i> (1)	<p>If <i>info</i>=0, on exit <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

Application Notes

For better performance, try using $lwork = n*(3*kd+3)$.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using `?syrd` instead of `?sytrd` on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. `?syrd` becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than `?sytrd`.

Avoid applying `?syrd` for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to Z is doubled compared to the traditional one-step reduction. In that case it is better to apply `?sytrd` and `?ormtr/?orgtr` to obtain tridiagonal form along with the orthogonal transformation matrix Q .

?herdb

Reduces a complex Hermitian matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

```
call cherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

```
call zherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- `mkl.fi`

Description

The routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*TQ^T$ and optionally multiplies matrix Z by Q , or simply forms Q .

This routine reduces a full symmetric matrix A to the banded symmetric matrix B , and then to the tridiagonal symmetric matrix T with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [Bischof00]). `?herdb` is functionally close to `?hetrd` routine but the tridiagonal form may differ from those obtained by `?hetrd`. Unlike `?hetrd`, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

`jobz` CHARACTER*1. Must be 'N' or 'V'.
 If `jobz = 'N'`, then only A is reduced to T .
 If `jobz = 'V'`, then A is reduced to T and A contains Q on exit.
 If `jobz = 'U'`, then A is reduced to T and Z contains Z^*Q on exit.

`uplo` CHARACTER*1. Must be 'U' or 'L'.
 If `uplo = 'U'`, a stores the upper triangular part of A .
 If `uplo = 'L'`, a stores the lower triangular part of A .

`n` INTEGER. The order of the matrix A ($n \geq 0$).

`kd` INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$, $kd \leq n-1$).

`a,z, work` COMPLEX for `cherdb`.
 DOUBLE COMPLEX for `zherdb`.
 $a(lda,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by `uplo`.
 The second dimension of a must be at least $\max(1, n)$.
 $z(ldz,*)$, the second dimension of z must be at least $\max(1, n)$.
 If `jobz = 'U'`, then the matrix z is multiplied by Q .

If $jobz = 'N'$ or $'V'$, then z is not referenced.

$work(lwork)$ is a workspace array.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldz INTEGER. The leading dimension of z ; at least $\max(1, n)$. Not referenced if $jobz = 'N'$

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq (2kd+1)n+kd$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a If $jobz = 'V'$, then overwritten by Q matrix.

If $jobz = 'N'$ or $'U'$, then overwritten by the banded matrix B and details of the unitary matrix Q_B to reduce A to B as specified by $uplo$.

z On exit,
if $jobz = 'U'$, then the matrix z is overwritten by Z^*Q .

If $jobz = 'N'$ or $'V'$, then z is not referenced.

d, e REAL for `cherdb`.

DOUBLE PRECISION for `zherdb`.

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

$tau(*)$ stores further details of the orthogonal matrix Q .

The dimension of tau must be at least $\max(1, n-kd-1)$.

tau COMPLEX for `cherdb`.

DOUBLE COMPLEX for `zherdb`.

Array, size at least $\max(1, n-1)$

Stores further details of the unitary matrix Q_B . The dimension of tau must be at least $\max(1, n-kd-1)$.

$work(1)$ If $info=0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

For better performance, try using $lwork = n*(3*kd+3)$.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using kd equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using `?herbdb` instead of `?hetrd` on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. `?herbdb` becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than `?hetrd`.

Avoid applying `?herbdb` for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to Z is doubled compared to the traditional one-step reduction. In that case it is better to apply `?hetrd` and `?unmtr/?ungtr` to obtain tridiagonal form along with the unitary transformation matrix Q .

?orgtr

Generates the real orthogonal matrix Q determined by `?sytrd`.

Syntax

```
call sorgtr(uplo, n, a, lda, tau, work, lwork, info)
call dorgtr(uplo, n, a, lda, tau, work, lwork, info)
call orgtr(a, tau [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by `sytrd` when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T Q$. Use this routine after a call to `?sytrd`.

Input Parameters

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Use the same $uplo$ as supplied to <code>?sytrd</code> .
n	INTEGER. The order of the matrix Q ($n \geq 0$).
$a, tau, work$	REAL for <code>sorgtr</code>

DOUBLE PRECISION for `dorgtr`.

Arrays:

$a(lda,*)$ is the array a as returned by `?sytrd`.

The second dimension of a must be at least $\max(1, n)$.

$tau(*)$ is the array tau as returned by `?sytrd`.

The size of tau must be at least $\max(1, n-1)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a Overwritten by the orthogonal matrix Q .

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orgtr` interface are the following:

a Holds the matrix A of size (n,n) .

tau Holds the vector of length $(n-1)$.

$uplo$ Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = (n-1) * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [ungtr](#).

?ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sytrd.

Syntax

```
call sormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call dormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call ormtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by [sytrd](#) when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T Q$. Use this routine after a call to [?sytrd](#).

Depending on the parameters $side$ and $trans$, the routine can form one of the matrix products $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

$side$	CHARACTER*1. Must be either 'L' or 'R'. If $side = 'L'$, Q or Q^T is applied to C from the left. If $side = 'R'$, Q or Q^T is applied to C from the right.
$uplo$	CHARACTER*1. Must be 'U' or 'L'. Use the same $uplo$ as supplied to ?sytrd .
$trans$	CHARACTER*1. Must be either 'N' or 'T'. If $trans = 'N'$, the routine multiplies C by Q . If $trans = 'T'$, the routine multiplies C by Q^T .

<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>a, c, tau, work</i>	<p>REAL for <code>sormtr</code> DOUBLE PRECISION for <code>dormtr</code></p> <p><i>a</i>(<i>lda</i>,*) and <i>tau</i> are the arrays returned by <code>?sytrd</code>. The second dimension of <i>a</i> must be at least $\max(1, r)$. The size of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i>(<i>ldc</i>,*) contains the matrix C. The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if <i>side</i> = 'L'; $lwork \geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^T * C$, $C * Q$, or $C * Q^T$ (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormtr` interface are the following:

<i>a</i>	<p>Holds the matrix A of size (<i>r</i>,<i>r</i>). $r = m$ if <i>side</i> = 'L'.</p>
----------	---

	$r = n$ if $side = 'R'$.
τ	Holds the vector of length $(r-1)$.
c	Holds the matrix C of size (m,n) .
$side$	Must be 'L' or 'R'. The default value is 'L'.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$trans$	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ for $side = 'L'$, or $lwork = m*blocksize$ for $side = 'R'$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately $2*m^2*n$, if $side = 'L'$, or $2*n^2*m$, if $side = 'R'$.

The complex counterpart of this routine is [unmtr](#).

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

```
call chetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call zhetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call hetrd(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*T^*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with Q in this representation. (They are described later in this section.)

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>a, work</i>	<p>COMPLEX for <code>chetrd</code></p> <p>DOUBLE COMPLEX for <code>zhetrtd</code>.</p> <p><i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the matrix <i>A</i>, as specified by <i>uplo</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>A</i> is not referenced.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of <i>a</i>; at least $\max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array ($lwork \geq n$).</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	<p>On exit,</p> <p>if <i>uplo</i> = 'U', the diagonal and first superdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i>, and the elements above the first superdiagonal, with the array <i>tau</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors;</p> <p>if <i>uplo</i> = 'L', the diagonal and first subdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i>, and the elements below the first subdiagonal, with the array <i>tau</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors.</p>
<i>d, e</i>	<p>REAL for <code>chetrd</code></p> <p>DOUBLE PRECISION for <code>zhetrtd</code>.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the diagonal elements of the matrix <i>T</i>.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the off-diagonal elements of <i>T</i>.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n-1)$.</p>

<i>tau</i>	COMPLEX for <code>chetrdDOUBLE</code> COMPLEX for <code>zhetr</code> . Array, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hetrd` interface are the following:

<i>a</i>	Holds the matrix A of size (n,n) .
<i>tau</i>	Holds the vector of length $(n-1)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3) n^3$.

After calling this routine, you can call the following:

<code>ungtr</code>	to form the computed matrix Q explicitly
<code>unmtr</code>	to multiply a complex matrix by Q .

The real counterpart of this routine is [?sytrd](#).

?ungtr

Generates the complex unitary matrix Q determined by ?hetrd.

Syntax

```
call cungrtr(uplo, n, a, lda, tau, work, lwork, info)
```

```
call zungtr(uplo, n, a, lda, tau, work, lwork, info)
```

```
call ungrtr(a, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by [hetrd](#) when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to ?hetrd.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hetrd.
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>a, tau, work</i>	COMPLEX for cungrtr DOUBLE COMPLEX for zungtr. Arrays: <i>a(lda,*)</i> is the array <i>a</i> as returned by ?hetrd. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>tau(*)</i> is the array <i>tau</i> as returned by ?hetrd. The dimension of <i>tau</i> must be at least $\max(1, n-1)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($lwork \geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the unitary matrix Q .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ungtr` interface are the following:

a Holds the matrix *A* of size (*n*,*n*).
tau Holds the vector of length (*n*-1).
uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using $lwork = (n-1)*blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [orgtr](#).

?unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by ?hetrd.

Syntax

```
call cunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call zunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call unmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix Q formed by [hetrd](#) when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to [?hetrd](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H^*C , C^*Q , or C^*Q^H (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If $side = 'L'$, Q or Q^H is applied to C from the left. If $side = 'R'$, Q or Q^H is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hetrd .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If $trans = 'N'$, the routine multiplies C by Q . If $trans = 'C'$, the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	COMPLEX for cunmtr DOUBLE COMPLEX for zunmtr . <i>a(lda,*)</i> and <i>tau</i> are the arrays returned by ?hetrd . The second dimension of <i>a</i> must be at least $\max(1, r)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c ldc,*)</i> contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if $side = 'L'$; $lwork \geq \max(1, m)$ if $side = 'R'$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmtr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>r</i>). $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r</i> -1).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, try using $lwork = n*blocksize$ (for *side* = 'L') or $lwork = m*blocksize$ (for *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is [ormtr](#).

?sptrd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

```
call ssptrd(uplo, n, ap, d, e, tau, info)
call dsptrd(uplo, n, ap, d, e, tau, info)
call sptrd(ap, tau [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a packed real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^*T^*Q^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation. See *Application Notes* below for details.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i>	REAL for <i>ssptrd</i> DOUBLE PRECISION for <i>dsptrd</i> . Array, size at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by <i>uplo</i>) in the packed form described in Matrix Storage Schemes .

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q , as specified by <i>uplo</i> .
<i>d, e, tau</i>	REAL for <i>ssptrd</i> DOUBLE PRECISION for <i>dsptrd</i> . Arrays: <i>d</i> (*) contains the diagonal elements of the matrix T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of T . The dimension of <i>e</i> must be at least $\max(1, n-1)$.

$\tau(*)$ Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the matrix Q in a product of $n-1$ reflectors.

The dimension of τ must be at least $\max(1, n-1)$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sptrd` interface are the following:

ap	Holds the array A of size $(n*(n+1)/2)$.
τ	Holds the vector with the number of elements $n-1$.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The matrix Q is represented as a product of $n-1$ *elementary reflectors*, as follows :

- If $uplo = 'U'$, $Q = H(n-1) \dots H(2)H(1)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T$$

where τ is a real scalar and v is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$.

On exit, τ is stored in $\tau(i)$, and $v(1:i-1)$ is stored in AP , overwriting $A(1:i-1, i+1)$.

- If $uplo = 'L'$, $Q = H(1)H(2) \dots H(n-1)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T$$

where τ is a real scalar and v is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$.

On exit, τ is stored in $\tau(i)$, and $v(i+2:n)$ is stored in AP , overwriting $A(i+2:n, i)$.

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

<code>opgtr</code>	to form the computed matrix Q explicitly
<code>opmtr</code>	to multiply a real matrix by Q .

The complex counterpart of this routine is [hpstrd](#).

?opgtr

Generates the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
call sopgtr(uplo, n, ap, tau, q, ldq, work, info)
call dopgtr(uplo, n, ap, tau, q, ldq, work, info)
call opgtr(ap, tau, q [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by [sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^*T^*Q^T$. Use this routine after a call to ?sptrd.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>ap, tau</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Arrays <i>ap</i> and <i>tau</i> , as returned by ?sptrd. The size of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The size of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> ; at least $\max(1, n)$.
<i>work</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Workspace array, size at least $\max(1, n-1)$.

Output Parameters

<i>q</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Array, size (<i>ldq</i> ,*) . Contains the computed matrix Q . The second dimension of <i>q</i> must be at least $\max(1, n)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `opgtr` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>tau</code>	Holds the vector with the number of elements $n - 1$.
<code>q</code>	Holds the matrix Q of size (n,n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [upgtr](#).

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
call sopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call opmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by [sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^T T Q$. Use this routine after a call to `sptrd`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^T C$, $Q^T C Q$, or $C Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If `side = 'L'`, $r = m$; if `side = 'R'`, $r = n$.

<code>side</code>	CHARACTER*1. Must be either 'L' or 'R'. If <code>side = 'L'</code> , Q or Q^T is applied to C from the left. If <code>side = 'R'</code> , Q or Q^T is applied to C from the right.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.

	Use the same <i>uplo</i> as supplied to <code>?sptd</code> .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies <i>C</i> by <i>Q</i> . If <i>trans</i> = 'T', the routine multiplies <i>C</i> by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix <i>C</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>C</i> ($n \geq 0$).
<i>ap, tau, c, work</i>	REAL for <code>sopmtr</code> DOUBLE PRECISION for <code>dopmtr</code> . <i>ap</i> and <i>tau</i> are the arrays returned by <code>?sptd</code> . The dimension of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c(ldc,*)</i> contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ if <i>side</i> = 'L'; $\max(1, m)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by <i>side</i> and <i>trans</i>).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `opmtr` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(r*(r+1)/2)$, where $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector with the number of elements $r - 1$.
<i>c</i>	Holds the matrix <i>C</i> of size (m,n) .

<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = 'L', or $2*n^2*m$ if *side* = 'R'.

The complex counterpart of this routine is [upmtr](#).

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

```
call chptrd(uplo, n, ap, d, e, tau, info)
call zhptrd(uplo, n, ap, d, e, tau, info)
call hptrd(ap, tau [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*T^*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i>	COMPLEX for <code>chptrd</code> DOUBLE COMPLEX for <code>zhptrd</code> . Array, size at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by <i>uplo</i>) in the packed form described in " Matrix Storage Schemes ".

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix T and details of the unitary matrix Q , as specified by <i>uplo</i> .
<i>d, e</i>	REAL for <code>chptrd</code> DOUBLE PRECISION for <code>zhptrd</code> . Arrays: <i>d</i> (*) contains the diagonal elements of the matrix T . The size of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of T . The size of <i>e</i> must be at least $\max(1, n-1)$.
<i>tau</i>	COMPLEX for <code>chptrd</code> DOUBLE COMPLEX for <code>zhptrd</code> . Array, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of reflectors.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hptrd` interface are the following:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>tau</i>	Holds the vector with the number of elements $n - 1$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3) n^3$.

After calling this routine, you can call the following:

<code>upgtr</code>	to form the computed matrix Q explicitly
<code>upmtr</code>	to multiply a complex matrix by Q .

The real counterpart of this routine is [sptrd](#).

?upgtr

Generates the complex unitary matrix Q determined by ?hptrd.

Syntax

```
call cupgtr(uplo, n, ap, tau, q, ldq, work, info)
```

```
call zupgtr(uplo, n, ap, tau, q, ldq, work, info)
```

```
call upgtr(ap, tau, q [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by [hptrd](#) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to [?hptrd](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hptrd.
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>ap, tau</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Arrays <i>ap</i> and <i>tau</i> , as returned by ?hptrd. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> ; at least $\max(1, n)$.
<i>work</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Workspace array, size at least $\max(1, n-1)$.

Output Parameters

<i>q</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zupgtr. Array, size (<i>ldq</i> ,*) . Contains the computed matrix Q . The second dimension of <i>q</i> must be at least $\max(1, n)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `upgtr` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>tau</code>	Holds the vector with the number of elements $n - 1$.
<code>q</code>	Holds the matrix Q of size (n,n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [opgtr](#).

?upmtr

Multiplies a complex matrix by the unitary matrix Q determined by ?hptrd.

Syntax

```
call cupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call upmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix formed by [hptrd](#) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^H T Q$. Use this routine after a call to [?hptrd](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^H C$, $Q C$, $C^H Q$, or $C Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If `side = 'L'`, $r = m$; if `side = 'R'`, $r = n$.

<code>side</code>	CHARACTER*1. Must be either 'L' or 'R'. If <code>side = 'L'</code> , Q or Q^H is applied to C from the left. If <code>side = 'R'</code> , Q or Q^H is applied to C from the right.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'.

	Use the same <i>uplo</i> as supplied to <code>?hptrd</code> .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies <i>C</i> by <i>Q</i> . If <i>trans</i> = 'T', the routine multiplies <i>C</i> by <i>Q^H</i> .
<i>m</i>	INTEGER. The number of rows in the matrix <i>C</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>C</i> ($n \geq 0$).
<i>ap, tau, c,</i>	COMPLEX for <code>cupmtr</code> DOUBLE COMPLEX for <code>zupmtr</code> . <i>ap</i> and <i>tau</i> are the arrays returned by <code>?hptrd</code> . The size of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The size of <i>tau</i> must be at least $\max(1, r-1)$. <i>c(ldc,*)</i> contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ if <i>side</i> = 'L'; $\max(1, m)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$.

Output Parameters

<i>c</i>	Overwritten by the product <i>Q</i> * <i>C</i> , <i>Q^H</i> * <i>C</i> , <i>C</i> * <i>Q</i> , or <i>C</i> * <i>Q^H</i> (as specified by <i>side</i> and <i>trans</i>).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `upmtr` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $r*(r+1)/2$, where $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector with the number of elements $n - 1$.
<i>c</i>	Holds the matrix <i>C</i> of size (m,n) .

<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is [opmtr](#).

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

```
call ssbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call sbtrd(ab[, q] [, vect] [, uplo] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q * T * Q^T$. The orthogonal matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

<i>vect</i>	CHARACTER*1. Must be 'V', 'N', or 'U'. If <i>vect</i> = 'V', the routine returns the explicit matrix Q . If <i>vect</i> = 'N', the routine does not return Q . If <i>vect</i> = 'U', the routine updates matrix X by forming $X * Q$.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).

ab, *q*, *work* REAL for *ssbtrd*
 DOUBLE PRECISION for *dsbtrd*.

ab(ldab,)* is an array containing either upper or lower triangular part of the matrix *A* (as specified by *uplo*) in band storage format.
 The second dimension of *ab* must be at least $\max(1, n)$.
q(ldq,)* is an array.
 If *vect* = 'U', the *q* array must contain an *n*-by-*n* matrix *X*.
 If *vect* = 'N' or 'V', the *q* parameter need not be set.
 The second dimension of *q* must be at least $\max(1, n)$.
work()* is a workspace array.
 The dimension of *work* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of *ab*; at least $kd+1$.

ldq INTEGER. The leading dimension of *q*. Constraints:
 $ldq \geq \max(1, n)$ if *vect* = 'V' or 'U';
 $ldq \geq 1$ if *vect* = 'N'.

Output Parameters

ab On exit, the diagonal elements of the array *ab* are overwritten by the diagonal elements of the tridiagonal matrix *T*. If $kd > 0$, the elements on the first superdiagonal (if *uplo* = 'U') or the first subdiagonal (if *uplo* = 'L') are overwritten by the off-diagonal elements of *T*. The rest of *ab* is overwritten by values generated during the reduction.

d, *e*, *q* REAL for *ssbtrd*
 DOUBLE PRECISION for *dsbtrd*.

Arrays:

d()* contains the diagonal elements of the matrix *T*.
 The size of *d* must be at least $\max(1, n)$.
e()* contains the off-diagonal elements of *T*.
 The size of *e* must be at least $\max(1, n-1)$.
q(ldq,)* is not referenced if *vect* = 'N'.
 If *vect* = 'V', *q* contains the *n*-by-*n* matrix *Q*.
 If *vect* = 'U', *q* contains the product *X*Q*.
 The second dimension of *q* must be:
 at least $\max(1, n)$ if *vect* = 'V';
 at least 1 if *vect* = 'N'.

info INTEGER.
 If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbtrd` interface are the following:

<code>ab</code>	Holds the array A of size $(kd+1, n)$.
<code>q</code>	Holds the matrix Q of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vect</code>	If omitted, this argument is restored based on the presence of argument q as follows: $vect = 'V'$, if q is present, $vect = 'N'$, if q is omitted. If present, $vect$ must be equal to 'V' or 'U' and the argument q must also be present. Note that there will be an error condition if $vect$ is present and q omitted.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n)*\epsilon*\|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $6n^2*kd$ if $vect = 'N'$, with $3n^3*(kd-1)/kd$ additional operations if $vect = 'V'$.

The complex counterpart of this routine is [hbtrd](#).

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

```
call chbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call hbtrd(ab [, q] [, vect] [, uplo] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

<i>vect</i>	<p>CHARACTER*1. Must be 'V', 'N', or 'U'.</p> <p>If <i>vect</i> = 'V', the routine returns the explicit matrix <i>Q</i>.</p> <p>If <i>vect</i> = 'N', the routine does not return <i>Q</i>.</p> <p>If <i>vect</i> = 'U', the routine updates matrix <i>X</i> by forming Q^*X.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	<p>INTEGER. The number of super- or sub-diagonals in <i>A</i></p> <p>($kd \geq 0$).</p>
<i>ab, work</i>	<p>COMPLEX for <i>chbtrd</i></p> <p>DOUBLE COMPLEX for <i>zhbtrd</i>.</p> <p><i>ab(ldab,*)</i> is an array containing either upper or lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>work(*)</i> is a workspace array.</p> <p>The dimension of <i>work</i> must be at least $\max(1, n)$.</p>
<i>q</i>	<p>COMPLEX for <i>chbtrd</i></p> <p>DOUBLE COMPLEX for <i>zhbtrd</i>.</p> <p><i>q(ldq,*)</i> is an array.</p> <p>If <i>vect</i> = 'U', the <i>q</i> array must contain an <i>n</i>-by-<i>n</i> matrix <i>X</i>.</p> <p>If <i>vect</i> = 'N' or 'V', the <i>q</i> parameter need not be set.'</p>
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; at least $kd+1$.
<i>ldq</i>	<p>INTEGER. The leading dimension of <i>q</i>. Constraints:</p> <p>$ldq \geq \max(1, n)$ if <i>vect</i> = 'V' or 'U';</p> <p>$ldq \geq 1$ if <i>vect</i> = 'N'.</p>

Output Parameters

<i>ab</i>	<p>On exit, the diagonal elements of the array <i>ab</i> are overwritten by the diagonal elements of the tridiagonal matrix <i>T</i>. If $kd > 0$, the elements on the first superdiagonal (if <i>uplo</i> = 'U') or the first subdiagonal (if <i>uplo</i> = 'L') are overwritten by the off-diagonal elements of <i>T</i>. The rest of <i>ab</i> is overwritten by values generated during the reduction.</p>
<i>d, e</i>	<p>REAL for <i>chbtrd</i></p> <p>DOUBLE PRECISION for <i>zhbtrd</i>.</p>

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

q

If $vect = 'N'$, q is not referenced.

If $vect = 'V'$, q contains the n -by- n matrix Q .

If $vect = 'U'$, q contains the product $X^* Q$.

The second dimension of q must be:

at least $\max(1, n)$ if $vect = 'V'$;

at least 1 if $vect = 'N'$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbtrd` interface are the following:

ab Holds the array A of size $(kd+1, n)$.

q Holds the matrix Q of size (n, n) .

$uplo$ Must be 'U' or 'L'. The default value is 'U'.

$vect$ If omitted, this argument is restored based on the presence of argument q as follows: $vect = 'V'$, if q is present, $vect = 'N'$, if q is omitted.
If present, $vect$ must be equal to 'V' or 'U' and the argument q must also be present. Note that there will be an error condition if $vect$ is present and q omitted.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $20n^2 * kd$ if $vect = 'N'$, with $10n^3 * (kd-1) / kd$ additional operations if $vect = 'V'$.

The real counterpart of this routine is [sbtrd](#).

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

```
call ssterf(n, d, e, info)
call dsterf(n, d, e, info)
call sterf(d, e [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues of a real symmetric tridiagonal matrix T (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call [steqr](#).

Input Parameters

n INTEGER. The order of the matrix T ($n \geq 0$).

d, e REAL for `ssterf`
DOUBLE PRECISION for `dsterf`.

Arrays:
d(*) contains the diagonal elements of T .
The dimension of *d* must be at least $\max(1, n)$.
e(*) contains the off-diagonal elements of T .
The dimension of *e* must be at least $\max(1, n-1)$.

Output Parameters

d The n eigenvalues in ascending order, unless *info* > 0.
See also *info*.

e On exit, the array is overwritten; see *info*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = *i*, the algorithm failed to find all the eigenvalues after $30n$ iterations:
i off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .
If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sterf` interface are the following:

<code>d</code>	Holds the vector of length n .
<code>e</code>	Holds the vector of length $(n-1)$.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If λ_j is an exact eigenvalue, and m_j is the corresponding computed value, then

$$|\mu_j - \lambda_j| \leq c(n) * \varepsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

```
call ssteqr(compz, n, d, e, z, ldz, work, info)
call dsteqr(compz, n, d, e, z, ldz, work, info)
call csteqr(compz, n, d, e, z, ldz, work, info)
call zsteqr(compz, n, d, e, z, ldz, work, info)
call rsteqr(d, e [,z] [,compz] [,info])
call steqr(d, e [,z] [,compz] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z * \Lambda * Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T : $A = Q * T * Q^H$. In this case, the spectral factorization is as follows: $A = Q * T * Q^H = (Q * Z) * \Lambda * (Q * Z)^H$. Before calling `?steqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd(<i>vect</i> ='V')	?hbtrd(<i>vect</i> ='V')

If you need eigenvalues only, it's more efficient to call [sterf](#). If T is positive-definite, [pteqr](#) can compute small eigenvalues more accurately than [?steqr](#).

To solve the problem by a single call, use one of the divide and conquer routines [stevd](#), [syevd](#), [spevd](#), or [sbevd](#) for real symmetric matrices or [heevd](#), [hpevd](#), or [hbevd](#) for complex Hermitian matrices.

Input Parameters

<i>compz</i>	<p>CHARACTER*1. Must be 'N' or 'I' or 'V'.</p> <p>If <i>compz</i> = 'N', the routine computes eigenvalues only.</p> <p>If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T.</p> <p>If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of the original symmetric matrix. On entry, z must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.</p>
<i>n</i>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<i>d, e, work</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Arrays:</p> <p>$d(*)$ contains the diagonal elements of T.</p> <p>The size of d must be at least $\max(1, n)$.</p> <p>$e(*)$ contains the off-diagonal elements of T.</p> <p>The size of e must be at least $\max(1, n-1)$.</p> <p>$work(*)$ is a workspace array.</p> <p>The size of $work$ must be:</p> <p>at least 1 if <i>compz</i> = 'N';</p> <p>at least $\max(1, 2*n-2)$ if <i>compz</i> = 'V' or 'I'.</p>
<i>z</i>	<p>REAL for <i>ssteqr</i></p> <p>DOUBLE PRECISION for <i>dsteqr</i></p> <p>COMPLEX for <i>csteqr</i></p> <p>DOUBLE COMPLEX for <i>zsteqr</i>.</p> <p>Array, size (<i>ldz</i>, *).</p> <p>If <i>compz</i> = 'N' or 'I', z need not be set.</p> <p>If <i>vect</i> = 'V', z must contain the orthogonal matrix used in the reduction to tridiagonal form.</p> <p>The second dimension of z must be:</p>

at least 1 if `compz = 'N'`;
 at least $\max(1, n)$ if `compz = 'V' or 'I'`.
`work` (`lwork`) is a workspace array.

`ldz` INTEGER. The leading dimension of `z`. Constraints:

$ldz \geq 1$ if `compz = 'N'`;
 $ldz \geq \max(1, n)$ if `compz = 'V' or 'I'`.

Output Parameters

`d` The n eigenvalues in ascending order, unless `info > 0`.

See also `info`.

`e` On exit, the array is overwritten; see `info`.

`z` If `info = 0`, contains the n -by- n matrix the columns of which are orthonormal eigenvectors (the i -th column corresponds to the i -th eigenvalue).

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = i`, the algorithm failed to find all the eigenvalues after $30n$ iterations: i off-diagonal elements have not converged to zero. On exit, `d` and `e` contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .

If `info = -i`, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `steqr` interface are the following:

`d` Holds the vector of length n .

`e` Holds the vector of length $(n-1)$.

`z` Holds the matrix Z of size (n,n) .

`compz` If omitted, this argument is restored based on the presence of argument `z` as follows:

`compz = 'I'`, if `z` is present,

`compz = 'N'`, if `z` is omitted.

If present, `compz` must be equal to `'I'` or `'V'` and the argument `z` must also be present. Note that there will be an error condition if `compz` is present and `z` omitted.

Note that two variants of Fortran 95 interface for `steqr` routine are needed because of an ambiguous choice between real and complex cases appear when `z` is omitted. Thus, the name `rsteqr` is used in real cases (single or double precision), and the name `steqr` is used in complex cases (single or double precision).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If λ_j is an exact eigenvalue, and μ_j is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \varepsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if `compz = 'N'`;

$7n^3$ (for complex flavors, $14n^3$) if `compz = 'V' or 'I'`.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call dstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call cstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

```
call zstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices `il:iu` for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable $L^*D^*L^T$ factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of T ,

- a.** Compute $T - \text{sigma} * I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of L and D cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes all eigenvalues in the half-open interval: (<i>vl</i>, <i>vu</i>].</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>n</i>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<i>d</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size (<i>n</i>).</p> <p>Contains <i>n</i> diagonal elements of the tridiagonal matrix T.</p>
<i>e</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, size <i>n</i>.</p> <p>Contains (<i>n</i>-1) off-diagonal elements of the tridiagonal matrix T in elements 1 to <i>n</i>-1 of <i>e</i>. <i>e</i>(<i>n</i>) need not be set on input, but is used internally as workspace.</p>
<i>vl, vu</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p>

	<p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $v_l < v_u$.</p> <p>If <i>range</i> = 'A' or 'I', v_l and v_u are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>.</p> <p>if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$;</p> <p>$ldz \geq 1$ otherwise.</p>
<i>nzc</i>	<p>INTEGER. The number of eigenvectors to be held in the array <i>z</i>.</p> <p>If <i>range</i> = 'A', then $nzc \geq \max(1, n)$;</p> <p>If <i>range</i> = 'V', then <i>nzc</i> is greater than or equal to the number of eigenvalues in the half-open interval: $(v_l, v_u]$.</p> <p>If <i>range</i> = 'I', then $nzc \geq iu - il + 1$.</p> <p>If <i>nzc</i> = -1, then a workspace query is assumed; the routine calculates the number of columns of the array <i>z</i> that are needed to hold the eigenvectors. This value is returned as the first entry of the array <i>z</i>, and no error message related to <i>nzc</i> is issued by the routine <code>xerbla</code>.</p>
<i>tryrac</i>	<p>LOGICAL.</p> <p>If <i>tryrac</i> = <code>.TRUE.</code> is true, it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can use possibly faster algorithms.</p> <p>If <i>tryrac</i> = <code>.FALSE.</code> is not true, the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.</p>
<i>work</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Workspace array, size (<i>lwork</i>).</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>,</p> <p>$lwork \geq \max(1, 18 * n)$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>.</p>

iwork INTEGER.
Workspace array, size (*liwork*).

liwork INTEGER.
The dimension of the array *iwork*.
 $lwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $lwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed.
If *liwork*=-1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by xerbla.

Output Parameters

d On exit, the array *d* is overwritten.

e On exit, the array *e* is overwritten.

m INTEGER.
The total number of eigenvalues found, $0 \leq m \leq n$.
If *range* = 'A', then $m=n$, and if *range* = 'I', then $m=iu-il+1$.

w REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, size (*n*).
The first *m* elements contain the selected eigenvalues in ascending order.

z REAL for sstemr
DOUBLE PRECISION for dstemr
COMPLEX for cstemr
DOUBLE COMPLEX for zstemr.
Array *z*(*ldz*,*), the second dimension of *z* must be at least $\max(1, m)$.
If *jobz* = 'V', and *info* = 0, then the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
If *jobz* = 'N', then *z* is not referenced.
Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and can be computed with a workspace query by setting *nzc*=-1, see description of the parameter *nzc*.

isuppz INTEGER.
Array, size ($2*\max(1, m)$).

The support of the eigenvectors in z , that is the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. $isuppz$ is only accessed when $jobz = 'V'$ and $n > 0$.

<code>tryrac</code>	On exit, TRUE. <code>tryrac</code> is set to .FALSE. if the matrix does not define its eigenvalues to high relative accuracy.
<code>work(1)</code>	On exit, if <code>info = 0</code> , then <code>work(1)</code> returns the optimal (and minimal) size of <code>lwork</code> .
<code>iwork(1)</code>	On exit, if <code>info = 0</code> , then <code>iwork(1)</code> returns the optimal size of <code>liwork</code> .
<code>info</code>	INTEGER. If = 0, the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value. If <code>info = 1</code> , internal error in ?larre occurred, if <code>info = 2</code> , internal error in ?larv occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call rstedc(d, e [,z] [,compz] [,info])
call stedc(d, e [,z] [,compz] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if `sytrd/hetrd` or `sptrd/hptrd` or `sbtrd/hbtrd` has been used to reduce this matrix to tridiagonal form.

See also `laed0`, `laed1`, `laed2`, `laed3`, `laed4`, `laed5`, `laed6`, `laed7`, `laed8`, `laed9`, and `laeda` used by this function.

Input Parameters

`compz` CHARACTER*1. Must be 'N' or 'I' or 'V'.
If `compz = 'N'`, the routine computes eigenvalues only.

If *compz* = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix.

If *compz* = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

n INTEGER. The order of the symmetric tridiagonal matrix ($n \geq 0$).

d, *e*, *rwork* REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays:

d(*) contains the diagonal elements of the tridiagonal matrix.

The dimension of *d* must be at least $\max(1, n)$.

e(*) contains the subdiagonal elements of the tridiagonal matrix.

The dimension of *e* must be at least $\max(1, n-1)$.

rwork is a workspace array, its dimension $\max(1, lrwork)$.

z, *work* REAL for *sstedc*

DOUBLE PRECISION for *dstedc*

COMPLEX for *csstedc*

DOUBLE COMPLEX for *zstedc*.

Arrays: *z*(*ldz*, *), *work*(*).

If *compz* = 'V', then, on entry, *z* must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

The second dimension of *z* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

ldz INTEGER. The leading dimension of *z*. Constraints:

$ldz \geq 1$ if *compz* = 'N';

$ldz \geq \max(1, n)$ if *compz* = 'V' or 'I'.

lwork INTEGER. The dimension of the array *work*.

For real functions *sstedc* and *dstedc*:

- If *compz* = 'N' or $n \leq 1$, *lwork* must be at least 1.
- If *compz* = 'V' and $n > 1$, *lwork* must be at least $1 + 3*n + 2*n*\log_2(n) + 4*n^2$, where $\log_2(n)$ is the smallest integer *k* such that $2^k \geq n$.
- If *compz* = 'I' and $n > 1$ then *lwork* must be at least $1 + 4*n + n^2$

Note that for *compz* = 'I' or 'V' and if *n* is less than or equal to the minimum divide size, usually 25, then *lwork* need only be $\max(1, 2*(n-1))$.

For complex functions *csstedc* and *zstedc*:

- If *compz* = 'N' or 'I', or $n \leq 1$, *lwork* must be at least 1.

- If $compz = 'V'$ and $n > 1$, $lwork$ must be at least n^2 .

Note that for $compz = 'V'$, and if n is less than or equal to the minimum divide size, usually 25, then $lwork$ need only be 1.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for the required value of $lwork$.

$lrwork$

INTEGER. The dimension of the array $rwork$ (used for complex flavors only).

If $compz = 'N'$, or $n \leq 1$, $lrwork$ must be at least 1.

If $compz = 'V'$ and $n > 1$, $lrwork$ must be at least $(1 + 3*n + 2*n*\log_2(n) + 4*n^2)$, where $\log_2(n)$ is the smallest integer k such that $2^k \geq n$.

If $compz = 'I'$ and $n > 1$, $lrwork$ must be at least $(1 + 4*n + 2*n^2)$.

Note that for $compz = 'V'$ or $'I'$, and if n is less than or equal to the minimum divide size, usually 25, then $lrwork$ need only be $\max(1, 2*(n-1))$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for the required value of $lrwork$.

$iwork$

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER. The dimension of the array $iwork$.

If $compz = 'N'$, or $n \leq 1$, $liwork$ must be at least 1.

If $compz = 'V'$ and $n > 1$, $liwork$ must be at least $(6 + 6*n + 5*n*\log_2(n))$, where $\log_2(n)$ is the smallest integer k such that $2^k \geq n$.

If $compz = 'I'$ and $n > 1$, $liwork$ must be at least $(3 + 5*n)$.

Note that for $compz = 'V'$ or $'I'$, and if n is less than or equal to the minimum divide size, usually 25, then $liwork$ need only be 1.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for the required value of $liwork$.

Output Parameters

d

The n eigenvalues in ascending order, unless $info \neq 0$.

See also *info*.

e

On exit, the array is overwritten; see *info*.

<i>z</i>	If <i>info</i> = 0, then if <i>compz</i> = 'V', <i>z</i> contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if <i>compz</i> = 'I', <i>z</i> contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If <i>compz</i> = 'N', <i>z</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>info</i> = 0, then <i>rwork(1)</i> returns the optimal <i>lrwork</i> (for complex flavors only).
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>i</i> /(<i>n</i> +1) through mod(<i>i</i> , <i>n</i> +1).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stedc` interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

Note that two variants of Fortran 95 interface for `stedc` routine are needed because of an ambiguous choice between real and complex cases appear when *z* and *work* are omitted. Thus, the name `rstedc` is used in real cases (single or double precision), and the name `stedc` is used in complex cases (single or double precision).

Application Notes

The required size of workspace arrays must be as follows.

For `sstedc/dstedc`:

If *compz* = 'N' or $n \leq 1$ then *lwork* must be at least 1.

If *compz* = 'V' and $n > 1$ then *lwork* must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n)$ = smallest integer *k* such that $2^k \geq n$.

If *compz* = 'I' and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If *compz* = 'N' or $n \leq 1$ then *liwork* must be at least 1.

If $compz = 'V'$ and $n > 1$ then $liwork$ must be at least $(6 + 6n + 5n \cdot \log_2 n)$.

If $compz = 'I'$ and $n > 1$ then $liwork$ must be at least $(3 + 5n)$.

For $cstedc/zstedc$:

If $compz = 'N'$ or $'I'$, or $n \leq 1$, $lwork$ must be at least 1.

If $compz = 'V'$ and $n > 1$, $lwork$ must be at least n^2 .

If $compz = 'N'$ or $n \leq 1$, $lrwork$ must be at least 1.

If $compz = 'V'$ and $n > 1$, $lrwork$ must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

If $compz = 'I'$ and $n > 1$, $lrwork$ must be at least $(1 + 4n + 2n^2)$.

The required value of $liwork$ for complex flavors is the same as for real flavors.

If $lwork$ (or $liwork$ or $lrwork$, if supplied) is equal to -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$, $rwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$, $lrwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call dstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call cstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call zstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call rstegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

```
call stegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T .

The spectrum may be computed either completely or partially by specifying either an interval $[vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

?stegr is a compatibility wrapper around the improved [stemr](#) routine. See its description for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

See also auxiliary [lasq2lasq5](#), [lasq6](#), used by this routine.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval:</p> $vl < w(i) \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>n</i>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<i>d, e, work</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the diagonal elements of T.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the subdiagonal elements of T in elements 1 to $n-1$; <i>e</i>(n) need not be set on input, but it is used as a workspace.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n)$.</p> <p><i>work</i>(<i>lwork</i>) is a workspace array.</p>
<i>vl, vu</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>. Constraints:</p>

$ldz \geq 1$ if $jobz = 'N'$;
 $ldz \geq \max(1, n)$ if $jobz = 'V'$.

lwork

INTEGER.

The dimension of the array *work*, $lwork \geq \max(1, 18*n)$ if $jobz = 'V'$, and $lwork \geq \max(1, 12*n)$ if $jobz = 'N'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* below for details.

iwork

INTEGER.

Workspace array, size (*liwork*).*liwork*

INTEGER.

The dimension of the array *iwork*, $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed..

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#). See *Application Notes* below for details.

Output Parameters

*d, e*On exit, *d* and *e* are overwritten.*m*

INTEGER. The total number of eigenvalues found,

 $0 \leq m \leq n$.If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.*w*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size at least $\max(1, n)$.The selected eigenvalues in ascending order, stored in $w(1)$ to $w(m)$.*z*REAL for *sstegr*DOUBLE PRECISION for *dstegr*COMPLEX for *cstegr*DOUBLE COMPLEX for *zstegr*.Array $z(ldz, *)$, the second dimension of *z* must be at least $\max(1, m)$.

If $jobz = 'V'$, and if $info = 0$, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

Note: if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used. Using $n = m$ is always safe.

isuppz

INTEGER.

Array, size at least $(2 * \max(1, m))$.

The support of the eigenvectors in z , that is the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. *isuppz* is only accessed when $jobz = 'V'$, and $n > 0$.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

iwork(1)

On exit, if $info = 0$, then *iwork(1)* returns the required minimal size of *liwork*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1x$, internal error in ?larre occurred,

If $info = 2x$, internal error in ?larrv occurred. Here the digit $x = \text{abs}(iinfo) < 10$, where *iinfo* is the non-zero error code returned by ?larre or ?larrv, respectively.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *stegr* interface are the following:

<i>d</i>	Holds the vector of length n .
<i>e</i>	Holds the vector of length n .
<i>w</i>	Holds the vector of length n .
<i>z</i>	Holds the matrix Z of size (n,m) .
<i>isuppz</i>	Holds the vector of length $(2*m)$.
<i>v1</i>	Default value for this argument is $v1 = - \text{HUGE}(v1)$ where $\text{HUGE}(a)$ means the largest machine number of the same precision as argument a .
<i>vu</i>	Default value for this argument is $vu = \text{HUGE}(v1)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.

<i>abstol</i>	Default value for this argument is <i>abstol</i> = 0.0 _{WP} .
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Note that two variants of Fortran 95 interface for *stegr* routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name *rstegr* is used in real cases (single or double precision), and the name *stegr* is used in complex cases (single or double precision).

Application Notes

?*stegr* works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of ?*stegr* may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?*pteqr*

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

```
call spteqr(compz, n, d, e, z, ldz, work, info)
call dpteqr(compz, n, d, e, z, ldz, work, info)
call cpteqr(compz, n, d, e, z, ldz, work, info)
call zpteqr(compz, n, d, e, z, ldz, work, info)
call rpteqr(d, e [,z] [,compz] [,info])
call pteqr(d, e [,z] [,compz] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z^* \Lambda^* Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T : $A = Q^* T^* Q^H$. In this case, the spectral factorization is as follows: $A = Q^* T^* Q^H = (QZ)^* \Lambda^* (QZ)^H$. Before calling `?pteqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<code>?sytrd</code> , <code>?orgtr</code>	<code>?hetrd</code> , <code>?ungtr</code>
packed storage	<code>?sptrd</code> , <code>?opgtr</code>	<code>?hptrd</code> , <code>?upgtr</code>
band storage	<code>?sbtrd(vect='V')</code>	<code>?hbtrd(vect='V')</code>

The routine first factorizes T as $L^* D^* L^H$ where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L^* D^{1/2}$ and calls `?bdsqr` to compute the singular values of B , which are the square roots of the eigenvalues of T .

Input Parameters

<code>compz</code>	<p>CHARACTER*1. Must be 'N' or 'I' or 'V'.</p> <p>If <code>compz = 'N'</code>, the routine computes eigenvalues only.</p> <p>If <code>compz = 'I'</code>, the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T.</p> <p>If <code>compz = 'V'</code>, the routine computes the eigenvalues and eigenvectors of A (and the array <code>z</code> must contain the matrix Q on entry).</p>
<code>n</code>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<code>d</code> , <code>e</code> , <code>work</code>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Arrays:</p> <p><code>d(*)</code> contains the diagonal elements of T. The size of <code>d</code> must be at least $\max(1, n)$.</p> <p><code>e(*)</code> contains the off-diagonal elements of T. The size of <code>e</code> must be at least $\max(1, n-1)$.</p> <p><code>work(*)</code> is a workspace array.</p>

The dimension of *work* must be:

at least 1 if *compz* = 'N';

at least $\max(1, 4*n-4)$ if *compz* = 'V' or 'I'.

z REAL for *spteqr*
 DOUBLE PRECISION for *dpteqr*
 COMPLEX for *cpteqr*
 DOUBLE COMPLEX for *zpteqr*.

Array, size (*ldz*,*)

If *compz* = 'N' or 'I', *z* need not be set.

If *compz* = 'V', *z* must contain the orthogonal matrix used in the reduction to tridiagonal form..

The second dimension of *z* must be:

at least 1 if *compz* = 'N';

at least $\max(1, n)$ if *compz* = 'V' or 'I'.

ldz INTEGER. The leading dimension of *z*. Constraints:

$ldz \geq 1$ if *compz* = 'N';

$ldz \geq \max(1, n)$ if *compz* = 'V' or 'I'.

Output Parameters

d The *n* eigenvalues in descending order, unless *info* > 0.
 See also *info*.

e On exit, the array is overwritten.

z If *info* = 0, contains an *n*-by-*n* matrix the columns of which are orthonormal eigenvectors. (The *i*-th column corresponds to the *i*-th eigenvalue.)

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, the leading minor of order *i* (and hence *T* itself) is not positive-definite.

If *info* = *n* + *i*, the algorithm for computing singular values failed to converge; *i* off-diagonal elements have not converged to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *pteqr* interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>compz</i>	<p>If omitted, this argument is restored based on the presence of argument <i>z</i> as follows:</p> <p><i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted.</p> <p>If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.</p>

Note that two variants of Fortran 95 interface for `pteqr` routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name `rpqr` is used in real cases (single or double precision), and the name `pteqr` is used in complex cases (single or double precision).

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * K * \lambda_i$$

where $c(n)$ is a modestly increasing function of n , ε is the machine precision, and $K = \frac{\| |DTD| \|_2}{\| (DTD)^{-1} \|_2}$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) \varepsilon K / \min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|).$$

Here $\min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if *compz* = 'N';

$6n^3$ (for complex flavors, $12n^3$) if *compz* = 'V' or 'I'.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

```
call sstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w, iblock,
isplit, work, iwork, info)
```

```
call dstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w, iblock,
isplit, work, iwork, info)
```

```
call stebz(d, e, m, nsplit, w, iblock, isplit [, order] [,vl] [,vu] [,il] [,iu]
[,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form $T = \text{diag}(T_1, T_2, \dots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to [stein](#) can also take advantage of the block structure.

See also [laebz](#).

Input Parameters

<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $v_l < w(i) \leq v_u$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>order</i>	<p>CHARACTER*1. Must be 'B' or 'E'.</p> <p>If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.</p> <p>If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.</p>
<i>n</i>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<i>v_l, v_u</i>	<p>REAL for <i>sstebz</i></p> <p>DOUBLE PRECISION for <i>dstebz</i>.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval:</p> $v_l < w(i) \leq v_u.$ <p>If <i>range</i> = 'A' or 'I', <i>v_l</i> and <i>v_u</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER. Constraint: $1 \leq il \leq iu \leq n$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues $w(i)$ such that $il \leq i \leq iu$ (assuming that the eigenvalues $w(i)$ are in ascending order).</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>REAL for <i>sstebz</i></p> <p>DOUBLE PRECISION for <i>dstebz</i>.</p> <p>The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i>.</p> <p>If $abstol \leq 0.0$, then the tolerance is taken as $\text{eps} * T$, where <i>eps</i> is the machine precision, and T is the 1-norm of the matrix T.</p>
<i>d, e, work</i>	<p>REAL for <i>sstebz</i></p> <p>DOUBLE PRECISION for <i>dstebz</i>.</p> <p>Arrays:</p>

$d(*)$ contains the diagonal elements of T .
 The size of d must be at least $\max(1, n)$.
 $e(*)$ contains the off-diagonal elements of T .
 The size of e must be at least $\max(1, n-1)$.
 $work(*)$ is a workspace array.
 The dimension of $work$ must be at least $\max(1, 4n)$.

iwork INTEGER. Workspace.
 Array, size at least $\max(1, 3n)$.

Output Parameters

m INTEGER. The actual number of eigenvalues found.

nsplit INTEGER. The number of diagonal blocks detected in T .

w REAL for *sstebz*
 DOUBLE PRECISION for *dstebz*.
 Array, size at least $\max(1, n)$. The computed eigenvalues, stored in $w(1)$ to $w(m)$.

iblock, isplit INTEGER.
 Arrays, size at least $\max(1, n)$.
 A positive value $iblock(i)$ is the block number of the eigenvalue stored in $w(i)$ (see also *info*).
 The leading *nsplit* elements of *isplit* contain points at which T splits into blocks T_i as follows: the block T_1 contains rows/columns 1 to *isplit*(1); the block T_2 contains rows/columns *isplit*(1)+1 to *isplit*(2), and so on.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = 1, for *range* = 'A' or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; $iblock(j) < 0$ indicates that the eigenvalue stored in $w(j)$ failed to converge.
 If *info* = 2, for *range* = 'I', the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = 'A'.
 If *info* = 3:
 for *range* = 'A' or 'V', same as *info* = 1;
 for *range* = 'I', same as *info* = 2.
 If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stebz` interface are the following:

<code>d</code>	Holds the vector of length n .
<code>e</code>	Holds the vector of length $(n-1)$.
<code>w</code>	Holds the vector of length n .
<code>iblock</code>	Holds the vector of length n .
<code>isplit</code>	Holds the vector of length n .
<code>order</code>	Must be 'B' or 'E'. The default value is 'B'.
<code>vl</code>	Default value for this argument is $vl = -\text{HUGE}(vl)$ where $\text{HUGE}(a)$ means the largest machine number of the same precision as argument a .
<code>vu</code>	Default value for this argument is $vu = \text{HUGE}(vu)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this argument is $abstol = 0.0_WP$.
<code>range</code>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

The eigenvalues of T are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard QR method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

```
call sstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call dstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call cstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
```

```
call zstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call stein(d, e, w, iblock, isplit, z [,ifailv] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by `?stebz` with `order = 'B'`, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after `?stebz`, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix T .

If T has been formed by reduction of a full symmetric or Hermitian matrix A to tridiagonal form, you can transform eigenvectors of T to eigenvectors of A by calling `?ormtr` or `?opmtr` (for real flavors) or by calling `?unmtr` or `?upmtr` (for complex flavors).

Input Parameters

<code>n</code>	INTEGER. The order of the matrix T ($n \geq 0$).
<code>m</code>	INTEGER. The number of eigenvectors to be returned.
<code>d, e, w</code>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <code>d(*)</code> contains the diagonal elements of T . The size of <code>d</code> must be at least $\max(1, n)$. <code>e(*)</code> contains the sub-diagonal elements of T stored in elements 1 to $n-1$. The size of <code>e</code> must be at least $\max(1, n-1)$. <code>w(*)</code> contains the eigenvalues of T , stored in <code>w(1)</code> to <code>w(m)</code> (as returned by stebz). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2 , again in non-decreasing order, and so on. Constraint: if <code>iblock(i) = iblock(i+1)</code> , <code>w(i) ≤ w(i+1)</code> . The size of <code>w</code> must be at least $\max(1, n)$.
<code>iblock, isplit</code>	INTEGER. Arrays, size at least $\max(1, n)$. The arrays <code>iblock</code> and <code>isplit</code> , as returned by <code>?stebz</code> with <code>order = 'B'</code> . If you did not call <code>?stebz</code> with <code>order = 'B'</code> , set all elements of <code>iblock</code> to 1, and <code>isplit(1)</code> to n .
<code>ldz</code>	INTEGER. The leading dimension of the output array <code>z</code> ; <code>ldz ≥ max(1, n)</code> .
<code>work</code>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

Workspace array, size at least $\max(1, 5n)$.

iwork INTEGER.

Workspace array, size at least $\max(1, n)$.

Output Parameters

z REAL for `sstein`
DOUBLE PRECISION for `dstein`
COMPLEX for `cstein`
DOUBLE COMPLEX for `zstein`.

Array, size (*ldz*, *).

If *info* = 0, *z* contains an *n*-by-*n* matrix the columns of which are orthonormal eigenvectors. (The *i*-th column corresponds to the *i*th eigenvalue.)

ifailv INTEGER.

Array, size at least $\max(1, m)$.

If *info* = *i* > 0, the first *i* elements of *ifailv* contain the indices of any eigenvectors that failed to converge.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then *i* eigenvectors (as indicated by the parameter *ifailv*) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array *z*.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stein` interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length <i>n</i> .
<i>w</i>	Holds the vector of length <i>n</i> .
<i>iblock</i>	Holds the vector of length <i>n</i> .
<i>isplit</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>m</i>).
<i>ifailv</i>	Holds the vector of length (<i>m</i>).

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T+E_i$, where $\|E_i\|_2 = O(\epsilon) * \|T\|_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

```
call sdisna(job, m, n, d, sep, info)
call ddisna(job, m, n, d, sep, info)
call disna(d, sep [,job] [,minmn] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m -by- n matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the i -th computed vector is given by

$$?lamch('E') * (anorm / sep(i))$$

where $anorm = \|A\|_2 = \max(|d(j)|)$. $sep(i)$ is not allowed to be smaller than $slamch('E') * anorm$ in order to limit the size of the error bound.

?disna may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'E', 'L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed: <i>job</i> = 'E': for the eigenvectors of a symmetric/Hermitian matrix; <i>job</i> = 'L': for the left singular vectors of a general matrix; <i>job</i> = 'R': for the right singular vectors of a general matrix.
<i>m</i>	INTEGER. The number of rows of the matrix ($m \geq 0$).
<i>n</i>	INTEGER. If <i>job</i> = 'L', or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if <i>job</i> = 'E'.
<i>d</i>	REAL for <code>sdisna</code> DOUBLE PRECISION for <code>ddisna</code> .

Array, dimension at least $\max(1,m)$ if $job = 'E'$, and at least $\max(1, \min(m,n))$ if $job = 'L'$ or $'R'$.

This array must contain the eigenvalues (if $job = 'E'$) or singular values (if $job = 'L'$ or $'R'$) of the matrix, in either increasing or decreasing order.

If singular values, they must be non-negative.

Output Parameters

<i>sep</i>	REAL for <i>sdisna</i> DOUBLE PRECISION for <i>ddisna</i> . Array, dimension at least $\max(1,m)$ if $job = 'E'$, and at least $\max(1, \min(m,n))$ if $job = 'L'$ or $'R'$. The reciprocal condition numbers of the vectors.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *disna* interface are the following:

<i>d</i>	Holds the vector of length $\min(m,n)$.
<i>sep</i>	Holds the vector of length $\min(m,n)$.
<i>job</i>	Must be $'E'$, $'L'$, or $'R'$. The default value is $'E'$.
<i>minmn</i>	Indicates which of the values m or n is smaller. Must be either $'M'$ or $'N'$, the default is $'M'$. If $job = 'E'$, this argument is superfluous, If $job = 'L'$ or $'R'$, this argument is used by the routine.

Generalized Symmetric-Definite Eigenvalue Problems

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors z that satisfy one of these equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is an n -by- n symmetric or Hermitian matrix, and B is an n -by- n symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist n real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices A and B).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling LAPACK routines described earlier in this chapter (see [Symmetric Eigenvalue Problems](#)).

If $itype = 3$, the generalized eigenproblem is $B^*A*z = lambda*z$

for $uplo = 'U'$: $C = U^*A*U^T$, $z = U^T*y$;

for $uplo = 'L'$: $C = L^T*A*L$, $z = L*y$.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^T*U$.

If $uplo = 'L'$, the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form $B = L*L^T$.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, b

REAL for *ssygst*

DOUBLE PRECISION for *dsygst*.

Arrays:

a(lda,)* contains the upper or lower triangle of *A*.

The second dimension of *a* must be at least $\max(1, n)$.

b(l db,)* contains the Cholesky-factored matrix *B*:

$B = U^T*U$ or $B = L*L^T$ (as returned by *?potrf*).

The second dimension of *b* must be at least $\max(1, n)$.

lda

INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

Output Parameters

a

The upper or lower triangle of *A* is overwritten by the upper or lower triangle of *C*, as specified by the arguments *itype* and *uplo*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *sygst* interface are the following:

a

Holds the matrix *A* of size (n,n) .

b

Holds the matrix *B* of size (n,n) .

itype

Must be 1, 2, or 3. The default value is 1.

uplo

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $\text{itype} = 1$) or B (if $\text{itype} = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hegst

Reduces a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
call chegst(itype, uplo, n, a, lda, b, ldb, info)
call zhegst(itype, uplo, n, a, lda, b, ldb, info)
call hegst(a, b [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a complex Hermitian positive-definite generalized eigenvalue problem to standard form.

<i>itype</i>	Problem	Result
1	$A*x = \lambda*B*x$	A overwritten by $\text{inv}(U^H)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^H)$
2	$A*B*x = \lambda*x$	A overwritten by $U*A*U^H$ or L^H*A*L
3	$B*A*x = \lambda*x$	

Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

itype INTEGER. Must be 1 or 2 or 3.
 If $\text{itype} = 1$, the generalized eigenproblem is $A*z = \text{lambda}*B*z$
 for $\text{uplo} = 'U'$: $C = (U^H)^{-1}*A*U^{-1}$;
 for $\text{uplo} = 'L'$: $C = L^{-1}*A*(L^H)^{-1}$.
 If $\text{itype} = 2$, the generalized eigenproblem is $A*B*z = \text{lambda}*z$
 for $\text{uplo} = 'U'$: $C = U*A*U^H$;
 for $\text{uplo} = 'L'$: $C = L^H*A*L$.
 If $\text{itype} = 3$, the generalized eigenproblem is $B*A*z = \text{lambda}*z$
 for $\text{uplo} = 'U'$: $C = U*A*U^H$;
 for $\text{uplo} = 'L'$: $C = L^H*A*L$.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^H * U$.

If *uplo* = 'L', the array *a* stores the lower triangle of *A*; you must supply *B* in the factored form $B = L * L^H$.

<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b</i>	COMPLEX for <i>chegst</i> DOUBLE COMPLEX for <i>zhegst</i> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the upper or lower triangle of <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the Cholesky-factored matrix <i>B</i> : $B = U^H * U$ or $B = L * L^H$ (as returned by <i>?potrf</i>). The second dimension of <i>b</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The upper or lower triangle of <i>A</i> is overwritten by the upper or lower triangle of <i>C</i> , as specified by the arguments <i>itype</i> and <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hegst* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by B^{-1} (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```
call sspgst(itype, uplo, n, ap, bp, info)
call dspgst(itype, uplo, n, ap, bp, info)
call spgst(ap, bp [,itype] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces real symmetric-definite generalized eigenproblems

$$A*x = \lambda*B*x, A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x$$

to the standard form $C*y = \lambda*y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?pptrf to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A*z = \lambda*B*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^T)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$; for <i>uplo</i> = 'L': $C = \text{inv}(L)*A*\text{inv}(L^T)$, $z = \text{inv}(L^T)*y$. If <i>itype</i> = 2, the generalized eigenproblem is $A*B*z = \lambda*z$ for <i>uplo</i> = 'U': $C = U*A*U^T$, $z = \text{inv}(U)*y$; for <i>uplo</i> = 'L': $C = L^T*A*L$, $z = \text{inv}(L^T)*y$. If <i>itype</i> = 3, the generalized eigenproblem is $B*A*z = \lambda*z$ for <i>uplo</i> = 'U': $C = U*A*U^T$, $z = U^T*y$; for <i>uplo</i> = 'L': $C = L^T*A*L$, $z = L*y$.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A ; you must supply B in the factored form $B = U^T*U$. If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A ; you must supply B in the factored form $B = L*L^T$.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ap, bp</i>	REAL for sspgst DOUBLE PRECISION for dspgst.

Arrays:

ap (*) contains the packed upper or lower triangle of A .

The dimension of ap must be at least $\max(1, n*(n+1)/2)$.

bp (*) contains the packed Cholesky factor of B (as returned by `?pptrf` with the same $uplo$ value).

The dimension of bp must be at least $\max(1, n*(n+1)/2)$.

Output Parameters

ap The upper or lower triangle of A is overwritten by the upper or lower triangle of C , as specified by the arguments $itype$ and $uplo$.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spgst` interface are the following:

ap Holds the array A of size $(n*(n+1)/2)$.

bp Holds the array B of size $(n*(n+1)/2)$.

$itype$ Must be 1, 2, or 3. The default value is 1.

$uplo$ Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hpgst

Reduces a generalized eigenvalue problem with a Hermitian matrix to a standard eigenvalue problem using packed storage.

Syntax

```
call chpgst(itype, uplo, n, ap, bp, info)
call zhpgst(itype, uplo, n, ap, bp, info)
call hpgst(ap, bp [,itype] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces generalized eigenproblems with Hermitian matrices

$$A^*z = \lambda^*B^*z, A^*B^*z = \lambda^*z, \text{ or } B^*A^*z = \lambda^*z.$$

to standard eigenproblems $C^*y = \lambda^*y$, using packed matrix storage. Here A is a complex Hermitian matrix, and B is a complex Hermitian positive-definite matrix. Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization: $B = U^H*U$ or $B = L*L^H$.

Input Parameters

<i>itype</i>	<p>INTEGER. Must be 1 or 2 or 3.</p> <p>If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^H)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$; for <i>uplo</i> = 'L': $C = \text{inv}(L)*A*\text{inv}(L^H)$, $z = \text{inv}(L^H)*y$.</p> <p>If <i>itype</i> = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$ for <i>uplo</i> = 'U': $C = U^*A^*U^H$, $z = \text{inv}(U)*y$; for <i>uplo</i> = 'L': $C = L^H*A^*L$, $z = \text{inv}(L^H)*y$.</p> <p>If <i>itype</i> = 3, the generalized eigenproblem is $B^*A^*z = \lambda^*z$ for <i>uplo</i> = 'U': $C = U^*A^*U^H$, $z = U^H*y$; for <i>uplo</i> = 'L': $C = L^H*A^*L$, $z = L*y$.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A; you must supply B in the factored form $B = U^H*U$.</p> <p>If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A; you must supply B in the factored form $B = L*L^H$.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>ap</i> , <i>bp</i>	<p>COMPLEX for <code>chpgstDOUBLE</code> COMPLEX for <code>zhpgst</code>.</p> <p>Arrays:</p> <p><i>ap</i> (*) contains the packed upper or lower triangle of A. The dimension of <i>a</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i> (*) contains the packed Cholesky factor of B (as returned by <code>?pptrf</code> with the same <i>uplo</i> value). The dimension of <i>b</i> must be at least $\max(1, n*(n+1)/2)$.</p>

Output Parameters

<i>ap</i>	<p>The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments <i>itype</i> and <i>uplo</i>.</p>
<i>info</i>	<p>INTEGER.</p>

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpgst` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

```
call ssgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call dsbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call sbgst(ab, bb [,x] [,uplo] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

To reduce the real symmetric-definite generalized eigenproblem $A*z = \lambda*B*z$ to the standard form $C*y = \lambda*y$, where A , B and C are banded, this routine must be preceded by a call to `pbstf`, which computes the split Cholesky factorization of the positive-definite matrix B : $B=S^T*S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^T*A*X$, where $X = \text{inv}(S)*Q$ and Q is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , $X*z$ is an eigenvector of the original system.

Input Parameters

`vect` CHARACTER*1. Must be 'N' or 'V'.

	<p>If <i>vect</i> = 'N', then matrix <i>X</i> is not returned;</p> <p>If <i>vect</i> = 'V', then matrix <i>X</i> is returned.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($ka \geq kb \geq 0$).
<i>ab, bb, work</i>	<p>REAL for <i>ssbgst</i></p> <p>DOUBLE PRECISION for <i>dsbgst</i></p> <p><i>ab(ldab,*)</i> is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of the array <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>bb(lddb,*)</i> is an array containing the banded split Cholesky factor of <i>B</i> as specified by <i>uplo</i>, <i>n</i> and <i>kb</i> and returned by pbstf/pbstf.</p> <p>The second dimension of the array <i>bb</i> must be at least $\max(1, n)$.</p> <p><i>work(*)</i> is a workspace array, dimension at least $\max(1, 2*n)$</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>lddb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldx</i>	<p>The leading dimension of the output array <i>x</i>. Constraints: if <i>vect</i> = 'N', then $ldx \geq 1$;</p> <p>if <i>vect</i> = 'V', then $ldx \geq \max(1, n)$.</p>

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	<p>REAL for <i>ssbgst</i></p> <p>DOUBLE PRECISION for <i>dsbgst</i></p> <p>Array.</p> <p>If <i>vect</i> = 'V', then <i>x(ldx,*)</i> contains the <i>n</i>-by-<i>n</i> matrix $X = \text{inv}(S) * Q$.</p> <p>If <i>vect</i> = 'N', then <i>x</i> is not referenced.</p> <p>The second dimension of <i>x</i> must be:</p> <p>at least $\max(1, n)$, if <i>vect</i> = 'V';</p>

at least 1, if `vect = 'N'`.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbgst` interface are the following:

<code>ab</code>	Holds the array <i>A</i> of size $(ka+1, n)$.
<code>bb</code>	Holds the array <i>B</i> of size $(kb+1, n)$.
<code>x</code>	Holds the matrix <i>X</i> of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vect</code>	Restored based on the presence of the argument <i>x</i> as follows: <code>vect = 'V'</code> , if <i>x</i> is present, <code>vect = 'N'</code> , if <i>x</i> is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

If *ka* and *kb* are much less than *n* then the total number of floating-point operations is approximately $6n^2 \cdot kb$, when `vect = 'N'`. Additional $(3/2)n^3 \cdot (kb/ka)$ operations are required when `vect = 'V'`.

?hbgst

Reduces a complex Hermitian positive-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?

pbstf.

Syntax

```
call chbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
```

```
call zhbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
```

```
call hbgst(ab, bb [,x] [,uplo] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

To reduce the complex Hermitian positive-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*x = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to [pbstf/pbstf](#), which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^H * S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^H * A * X$, where $X = \text{inv}(S) * Q$, and Q is a unitary matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , $X * z$ is an eigenvector of the original system.

Input Parameters

<i>vect</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>vect</i> = 'N', then matrix X is not returned;</p> <p>If <i>vect</i> = 'V', then matrix X is returned.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A.</p> <p>If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A.</p>
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($ka \geq kb \geq 0$).
<i>ab, bb, work</i>	<p>COMPLEX for <code>chbgst</code> DOUBLE COMPLEX for <code>zhbgst</code></p> <p><i>ab</i>(<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of the array <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>bb</i>(<i>lddb</i>,*) is an array containing the banded split Cholesky factor of B as specified by <i>uplo</i>, n and <i>kb</i> and returned by pbstf/pbstf.</p> <p>The second dimension of the array <i>bb</i> must be at least $\max(1, n)$.</p> <p><i>work</i>(*) is a workspace array, dimension at least $\max(1, n)$</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>lddb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldx</i>	<p>The leading dimension of the output array x. Constraints:</p> <p>if <i>vect</i> = 'N', then $ldx \geq 1$;</p> <p>if <i>vect</i> = 'V', then $ldx \geq \max(1, n)$.</p>
<i>rwork</i>	<p>REAL for <code>chbgst</code></p> <p>DOUBLE PRECISION for <code>zhbgst</code></p> <p>Workspace array, dimension at least $\max(1, n)$</p>

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	COMPLEX for <i>chbgst</i> DOUBLE COMPLEX for <i>zhbgst</i> Array. If <i>vect</i> = 'V', then <i>x</i> (<i>ldx</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix $X = \text{inv}(S) * Q$. If <i>vect</i> = 'N', then <i>x</i> is not referenced. The second dimension of <i>x</i> must be: at least $\max(1, n)$, if <i>vect</i> = 'V'; at least 1, if <i>vect</i> = 'N'.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hbgst* interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size $(ka+1, n)$.
<i>bb</i>	Holds the array <i>B</i> of size $(kb+1, n)$.
<i>x</i>	Holds the matrix <i>X</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	Restored based on the presence of the argument <i>x</i> as follows: <i>vect</i> = 'V', if <i>x</i> is present, <i>vect</i> = 'N', if <i>x</i> is omitted.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $20n^2 * kb$, when *vect* = 'N'. Additional $5n^3 * (kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst.

Syntax

```
call spbstf(uplo, n, kb, bb, ldbb, info)
```

```
call dpbstf(uplo, n, kb, bb, ldbb, info)
call cpbstf(uplo, n, kb, bb, ldbb, info)
call zpbstf(uplo, n, kb, bb, ldbb, info)
call pbstf(bb [, uplo] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix B . It is to be used in conjunction with [sbgst/hbgst](#).

The factorization has the form $B = S^T * S$ (or $B = S^H * S$ for complex flavors), where S is a band matrix of the same bandwidth as B and the following structure: S is upper triangular in the first $(n+kb)/2$ rows and lower triangular in the remaining rows.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', *bb* stores the upper triangular part of B .
 If *uplo* = 'L', *bb* stores the lower triangular part of B .

n INTEGER. The order of the matrix B ($n \geq 0$).

kb INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).

bb REAL for `spbstf`
 DOUBLE PRECISION for `dpbstf`
 COMPLEX for `cpbstf`
 DOUBLE COMPLEX for `zpbstf`.
bb(ldb,)* is an array containing either upper or lower triangular part of the matrix B (as specified by *uplo*) in band storage format.
 The second dimension of the array *bb* must be at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *bb*; must be at least $kb+1$.

Output Parameters

bb On exit, this array is overwritten by the elements of the split Cholesky factor S .

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = i , then the factorization could not be completed, because the updated element b_{ij} would be the square root of a negative number; hence the matrix B is not positive-definite.
 If *info* = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `pbstf` interface are the following:

<code>bb</code>	Holds the array B of size $(kb+1, n)$.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed factor S is the exact factor of a perturbed matrix $B + E$, where

$$|E| \leq c(kb + 1)\epsilon \|S^H\| \|S\|, \quad |e_{ij}| \leq c(kb + 1)\epsilon \sqrt{b_{ii}b_{jj}}$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kb is much less than n .

After calling this routine, you can call `sbgst/hbgst` to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of A).
- Eigenvalues may be complex even for a real matrix A .
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z , then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z .

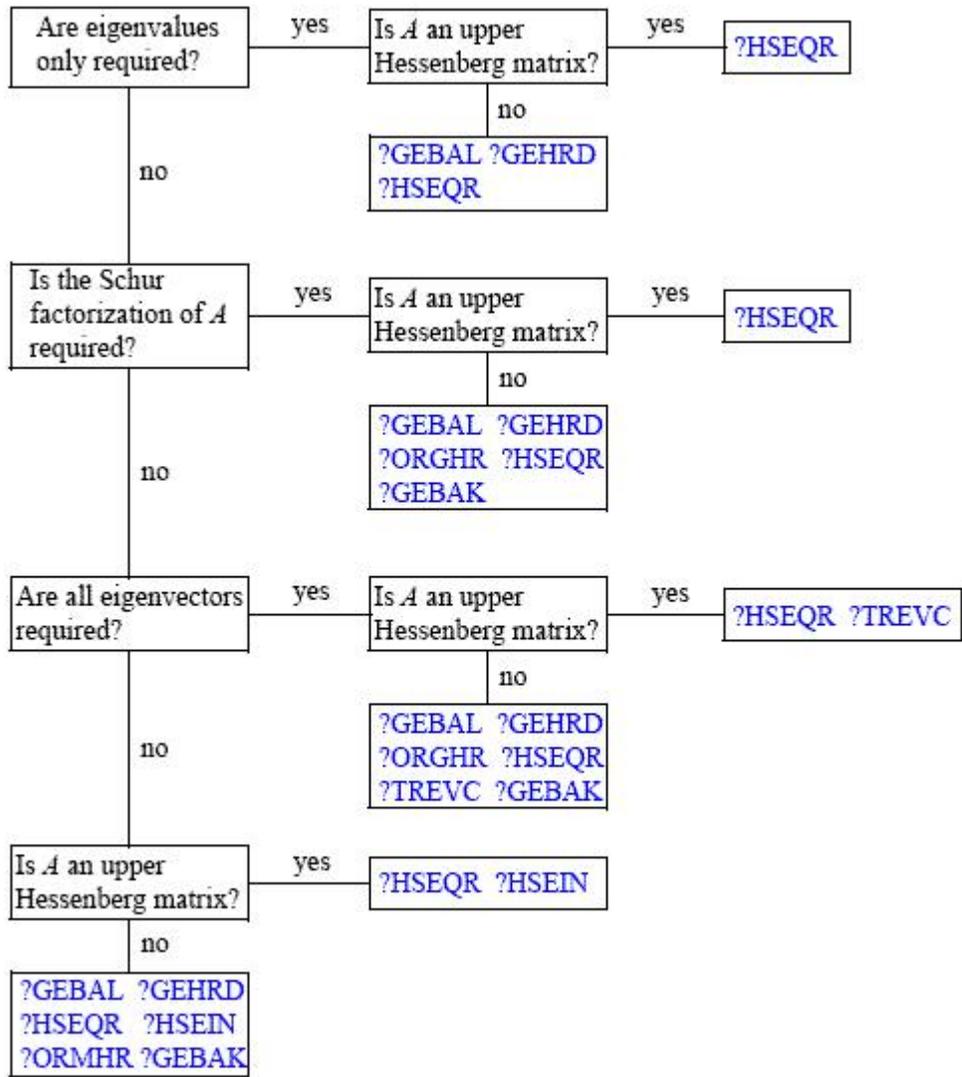
To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. [Table "Computational Routines for Solving Nonsymmetric Eigenvalue Problems"](#) lists LAPACK routines to reduce the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines to solve eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers. The corresponding routine names in the Fortran 95 interface are without the first symbol.

The decision tree in [Figure "Decision Tree: Real Nonsymmetric Eigenvalue Problems"](#) helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in [Figure "Decision Tree: Complex Non-Hermitian Eigenvalue Problems"](#).

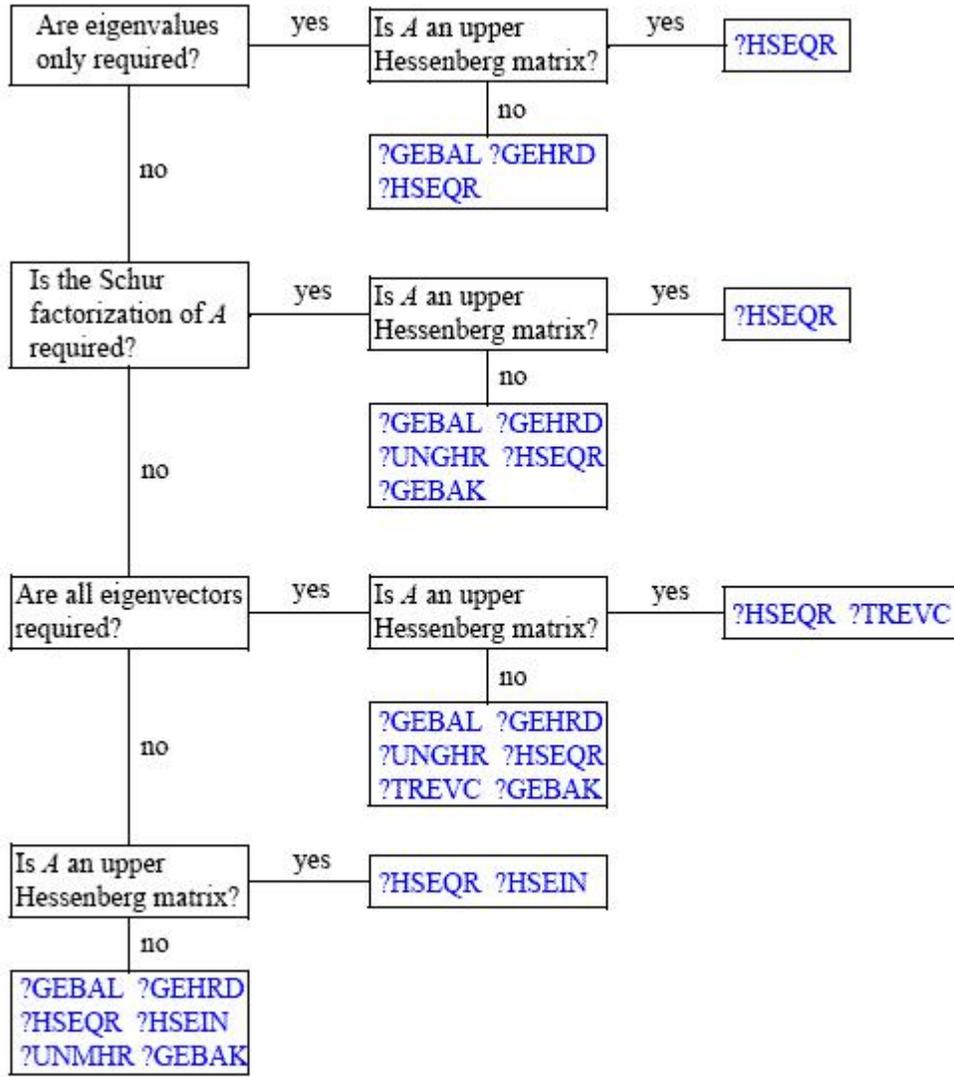
Computational Routines for Solving Nonsymmetric Eigenvalue Problems

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$?gehrd,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Decision Tree: Real Nonsymmetric Eigenvalue Problems



Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```

call sgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call cgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call gehrd(a [, tau] [,ilo] [,ihi] [,info])
  
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = Q^*H^*Q^H$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
ilo, ihi	INTEGER. If A is an output by <code>?gebal</code> , then ilo and ihi must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
$a, work$	REAL for <code>sgehrd</code> DOUBLE PRECISION for <code>dgehrd</code> COMPLEX for <code>cgehrd</code> DOUBLE COMPLEX for <code>zgehrd</code> . Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ ($lwork$) is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, n)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.

Output Parameters

a	The elements on and above the subdiagonal contain the upper Hessenberg matrix H . The subdiagonal elements of H are real. The elements below the subdiagonal, with the array tau , represent the orthogonal matrix Q as a product of n elementary reflectors.
tau	REAL for <code>sgehrd</code> DOUBLE PRECISION for <code>dgehrd</code> COMPLEX for <code>cgehrd</code> DOUBLE COMPLEX for <code>zgehrd</code> . Array, size at least $\max(1, n-1)$. Contains scalars that define elementary reflectors for the matrix Q .

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gehrd` interface are the following:

a Holds the matrix *A* of size (*n*,*n*).

tau Holds the vector of length (*n*-1).

ilo Default value for this argument is *ilo* = 1.

ihi Default value for this argument is *ihi* = *n*.

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Hessenberg matrix *H* is exactly similar to a nearby matrix $A + E$, where $\|E\|_2 < c(n)\epsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3) * (ihi - ilo)^2 (2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd.

Syntax

```
call sorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call orghr(a, tau [,ilo] [,ihi] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine explicitly generates the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents the matrix Q as a product of ihi - ilo elementary reflectors. Here ilo and ihi are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

The matrix Q generated by `?orghr` has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

- n INTEGER. The order of the matrix Q ($n \geq 0$).
- ilo, ihi INTEGER. These must be the same parameters ilo and ihi , respectively, as supplied to `?gehrd`. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
- $a, tau, work$ REAL for `sorghr`
DOUBLE PRECISION for `dorghr`
Arrays: $a(lda,*)$ contains details of the vectors which define the elementary reflectors, as returned by `?gehrd`.
The second dimension of a must be at least $\max(1, n)$.
 $tau(*)$ contains further details of the elementary reflectors, as returned by `?gehrd`.
The dimension of tau must be at least $\max(1, n-1)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
- lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.
- $lwork$ INTEGER. The size of the $work$ array;
 $lwork \geq \max(1, ihi-ilo)$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).
See *Application Notes* for the suggested value of $lwork$.

Output Parameters

<i>a</i>	Overwritten by the <i>n</i> -by- <i>n</i> orthogonal matrix <i>Q</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `orghr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>tau</i>	Holds the vector of length (<i>n</i> -1).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .

Application Notes

For better performance, try using $lwork = (ihi - ilo) * blocksize$ where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi - ilo)^3$.

The complex counterpart of this routine is [unghr](#).

?ormhr

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.

Syntax

```
call sormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
```

```
call dormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call ormhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine multiplies a matrix C by the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents the matrix Q as a product of *ihi-ilo* elementary reflectors. Here ilo and ihi are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With `?ormhr`, you can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$, overwriting the result on C (which may be any real rectangular matrix).

A common application of `?ormhr` is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

<code>side</code>	CHARACTER*1. Must be 'L' or 'R'. If <code>side= 'L'</code> , then the routine forms Q^*C or Q^T*C . If <code>side= 'R'</code> , then the routine forms $C*Q$ or $C*Q^T$.
<code>trans</code>	CHARACTER*1. Must be 'N' or 'T'. If <code>trans= 'N'</code> , then Q is applied to C . If <code>trans= 'T'</code> , then Q^T is applied to C .
<code>m</code>	INTEGER. The number of rows in C ($m \geq 0$).
<code>n</code>	INTEGER. The number of columns in C ($n \geq 0$).
<code>ilo, ihi</code>	INTEGER. These must be the same parameters ilo and ihi , respectively, as supplied to <code>?gehrd</code> . If $m > 0$ and <code>side = 'L'</code> , then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <code>side = 'L'</code> , then $ilo = 1$ and $ihi = 0$. If $n > 0$ and <code>side = 'R'</code> , then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <code>side = 'R'</code> , then $ilo = 1$ and $ihi = 0$.
<code>a, tau, c, work</code>	REAL for <code>sormhr</code> DOUBLE PRECISION for <code>dormhr</code> Arrays: <code>a(lda,*)</code> contains details of the vectors which define the <i>elementary reflectors</i> , as returned by <code>?gehrd</code> . The second dimension of <code>a</code> must be at least $\max(1, m)$ if <code>side = 'L'</code> and at least $\max(1, n)$ if <code>side = 'R'</code> .

$\tau(*)$ contains further details of the *elementary reflectors*, as returned by `?gehrd`.

The dimension of τ must be at least $\max(1, m-1)$ if $side = 'L'$ and at least $\max(1, n-1)$ if $side = 'R'$.

$c(ldc,*)$ contains the m by n matrix C .

The second dimension of c must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$ if $side = 'L'$ and at least $\max(1, n)$ if $side = 'R'$.

ldc INTEGER. The leading dimension of c ; at least $\max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array.

If $side = 'L'$, $lwork \geq \max(1, n)$.

If $side = 'R'$, $lwork \geq \max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c C is overwritten by product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ as specified by $side$ and $trans$.

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ormhr` interface are the following:

a Holds the matrix A of size (r,r) .

$r = m$ if $side = 'L'$.

$r = n$ if $side = 'R'$.

τ Holds the vector of length $(r-1)$.

c Holds the matrix C of size (m,n) .

<i>ilo</i>	Default value for this argument is $ilo = 1$.
<i>ihi</i>	Default value for this argument is $ihi = n$.
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least $n \cdot \text{blocksize}$ if *side* = 'L' and at least $m \cdot \text{blocksize}$ if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is [unmhr](#).

?unghr

Generates the complex unitary matrix Q determined by ?gehrd.

Syntax

```
call cunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zunghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call unghr(a, tau [,ilo] [,ihi] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine is intended to be used following a call to `cgehrd/zgehrd`, which reduces a complex matrix *A* to upper Hessenberg form *H* by a unitary similarity transformation: $A = Q^*H^*Q^H$. `?gehrd` represents the matrix *Q* as a product of *ihi-ilo* elementary reflectors. Here *ilo* and *ihi* are values determined by `cgebal/zgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.

Use the routine [unghr](#) to generate *Q* explicitly as a square matrix. The matrix *Q* has the structure:



where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

n	INTEGER. The order of the matrix Q ($n \geq 0$).
ilo, ihi	INTEGER. These must be the same parameters ilo and ihi , respectively, as supplied to <code>?gehrd</code> . (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.)
$a, tau, work$	<p>COMPLEX for <code>cunghr</code> DOUBLE COMPLEX for <code>zunghr</code>.</p> <p>Arrays:</p> <p>$a(lda,*)$ contains details of the vectors which define the <i>elementary reflectors</i>, as returned by <code>?gehrd</code>.</p> <p>The second dimension of a must be at least $\max(1, n)$.</p> <p>$tau(*)$ contains further details of the <i>elementary reflectors</i>, as returned by <code>?gehrd</code>.</p> <p>The dimension of tau must be at least $\max(1, n-1)$.</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
$lwork$	<p>INTEGER. The size of the $work$ array;</p> <p>$lwork \geq \max(1, ihi-ilo)$.</p> <p>If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of $lwork$.</p>

Output Parameters

a	Overwritten by the n -by- n unitary matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unghr` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>ilo</code>	Default value for this argument is $ilo = 1$.
<code>ihi</code>	Default value for this argument is $ihi = n$.

Application Notes

For better performance, try using $lwork = (ihi-ilo)*blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is [orghr](#).

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

```
call cunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call zunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call unmhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to `cgehrd/zgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^H$, and represents the matrix Q as a product of *ihi-ilo elementary reflectors*. Here *ilo* and *ihi* are values determined by `cgebal/zgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With `?unmhr`, you can form one of the matrix products Q^*C , Q^H^*C , C^*Q , or C^*Q^H , overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms Q^*C or Q^H^*C . If <i>side</i> = 'R', then the routine forms C^*Q or C^*Q^H .
<i>trans</i>	CHARACTER*1. Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q is applied to C . If <i>trans</i> = 'T', then Q^H is applied to C .
<i>m</i>	INTEGER. The number of rows in C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <code>?gehrd</code> . If $m > 0$ and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <i>side</i> = 'L', then $ilo = 1$ and $ihi = 0$. If $n > 0$ and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <i>side</i> = 'R', then $ilo = 1$ and $ihi = 0$.
<i>a, tau, c, work</i>	COMPLEX for <code>cunmhr</code> DOUBLE COMPLEX for <code>zunmhr</code> . Arrays: <i>a(lda,*)</i> contains details of the vectors which define the elementary reflectors, as returned by <code>?gehrd</code> . The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>tau(*)</i> contains further details of the elementary reflectors, as returned by <code>?gehrd</code> . The dimension of <i>tau</i> must be at least $\max(1, m-1)$ if <i>side</i> = 'L' and at least $\max(1, n-1)$ if <i>side</i> = 'R'. <i>c ldc,*)</i> contains the m -by- n matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. If <i>side</i> = 'L', $lwork \geq \max(1, n)$. If <i>side</i> = 'R', $lwork \geq \max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	<i>C</i> is overwritten by Q^*C , or Q^H*C , or $C*Q^H$, or $C*Q$ as specified by <i>side</i> and <i>trans</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `unmhr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r</i> , <i>r</i>). $r = m$ if <i>side</i> = 'L'. $r = n$ if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r</i> -1).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

where P is a permutation matrix, and A'_{11} and A'_{33} are upper triangular. The diagonal elements of A'_{11} and A'_{33} are eigenvalues of A . The rest of the eigenvalues of A are the eigenvalues of the central diagonal block A'_{22} , in rows and columns ilo to ihi . Subsequent operations to compute the eigenvalues of A (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if $ilo > 1$ and $ihi < n$.

If no suitable permutation exists (as is often the case), the routine sets $ilo = 1$ and $ihi = n$, and A'_{22} is the whole of A .

(2) The routine applies a diagonal similarity transformation to A' , to make the rows and columns of A'_{22} as close in norm as possible:

$$\dots, \begin{matrix} \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{matrix}, \begin{matrix} \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{matrix}, \dots, \dots, \begin{matrix} \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{matrix}, \dots$$

This scaling can reduce the norm of the matrix (that is, $\|A'_{22}\| < \|A'_{22}\|$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

- job* CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'.
 If *job* = 'N', then A is neither permuted nor scaled (but *ilo*, *ihi*, and *scale* get their values).
 If *job* = 'P', then A is permuted but not scaled.
 If *job* = 'S', then A is scaled but not permuted.
 If *job* = 'B', then A is both scaled and permuted.
- n* INTEGER. The order of the matrix A ($n \geq 0$).
- a* REAL for sgebal
 DOUBLE PRECISION for dgebal
 COMPLEX for cgebal
 DOUBLE COMPLEX for zgebal.
 Array $a(lda,*)$ contains the matrix A .
 The second dimension of a must be at least $\max(1, n)$. a is not referenced if *job* = 'N'.
- lda* INTEGER. The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

- a* Overwritten by the balanced matrix (a is not referenced if *job* = 'N').
- ilo, ihi* INTEGER. The values *ilo* and *ihi* such that on exit $a(i,j)$ is zero if $i > j$ and $1 \leq j < ilo$ or $ihi < j \leq n$.
 If *job* = 'N' or 'S', then $ilo = 1$ and $ihi = n$.

scale REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors
 Array, size at least $\max(1, n)$.
 Contains details of the permutations and scaling factors.
 More precisely, if p_j is the index of the row and column interchanged with row and column j , and d_j is the scaling factor used to balance row and column j , then
 $scale(j) = p_j$ for $j = 1, 2, \dots, ilo-1, ihi+1, \dots, n$;
 $scale(j) = d_j$ for $j = ilo, ilo + 1, \dots, ihi$.
 The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gebak` interface are the following:

a Holds the matrix A of size (n,n) .
scale Holds the vector of length n .
ilo Default value for this argument is $ilo = 1$.
ihi Default value for this argument is $ihi = n$.
job Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix A is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix A'' and hence you must call `gebak` to transform them back to eigenvectors of A .

If the Schur vectors of A are required, do not call this routine with $job = 'S'$ or $'B'$, because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with $job = 'P'$, then any Schur vectors computed subsequently are Schur vectors of the matrix A'' , and you need to call `gebak` (with $side = 'R'$) to transform them back to Schur vectors of A .

The total number of floating-point operations is proportional to n^2 .

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

```
call sgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
```

```
call dgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
```

```
call cgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call zgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call gebak(v, scale [,ilo] [,ihi] [,job] [,side] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine is intended to be used after a matrix A has been balanced by a call to `?gebal`, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see `gebal`. The balanced matrix A'' is obtained as $A'' = D*P*A*P^T*inv(D)$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if x is a right eigenvector of A'' , then $P^T*inv(D)*x$ is a right eigenvector of A ; if y is a left eigenvector of A'' , then P^T*D*y is a left eigenvector of A .

Input Parameters

<i>job</i>	CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'. The same parameter <i>job</i> as supplied to <code>?gebal</code> .
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then left eigenvectors are transformed. If <i>side</i> = 'R', then right eigenvectors are transformed.
<i>n</i>	INTEGER. The number of rows of the matrix of eigenvectors ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. The values <i>ilo</i> and <i>ihi</i> , as returned by <code>?gebal</code> . (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then $ilo = 1$ and $ihi = 0$.)
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, size at least $\max(1, n)$. Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by <code>?gebal</code> .
<i>m</i>	INTEGER. The number of columns of the matrix of eigenvectors ($m \geq 0$).
<i>v</i>	REAL for <code>sgebak</code> DOUBLE PRECISION for <code>dgebak</code> COMPLEX for <code>cgebak</code> DOUBLE COMPLEX for <code>zgebak</code> . Arrays: <i>v</i> (<i>ldv</i> ,*) contains the matrix of left or right eigenvectors to be transformed. The second dimension of <i>v</i> must be at least $\max(1, m)$.
<i>ldv</i>	INTEGER. The leading dimension of <i>v</i> ; at least $\max(1, n)$.

Output Parameters

<code>v</code>	Overwritten by the transformed eigenvectors.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gebak` interface are the following:

<code>v</code>	Holds the matrix V of size (n,m) .
<code>scale</code>	Holds the vector of length n .
<code>ilo</code>	Default value for this argument is <code>ilo = 1</code> .
<code>ihi</code>	Default value for this argument is <code>ihi = n</code> .
<code>job</code>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to $m*n$.

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
call shseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call dhseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call chseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call hseqr(h, wr, wi [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
call hseqr(h, w [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H : $H = Z^* T^* Z^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i .

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H :

$$A = Q^* H^* Q^H, \text{ where } Q \text{ is unitary (orthogonal for real flavors);}$$

$$A = (QZ)^* T^* (QZ)^H.$$

In this case, after reducing A to Hessenberg form by `gehrd`, call `orghr` to form Q explicitly and then pass Q to `?hseqr` with `compz = 'V'`.

You can also call `gebal` to balance the original matrix before reducing it to Hessenberg form by `?hseqr`, so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns ilo to ihi) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of ilo and ihi can be supplied to `?hseqr` directly. Also, after calling this routine you must call `gebak` to permute the Schur vectors of the balanced matrix to those of the original matrix.

If `?gebal` has not been called, however, then ilo must be set to 1 and ihi to n . Note that if the Schur factorization of A is required, `?gebal` must not be called with `job = 'S'` or `'B'`, because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

`?hseqr` uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that $\|z_i\|_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ± 1).

Input Parameters

<code>job</code>	CHARACTER*1. Must be 'E' or 'S'. If <code>job = 'E'</code> , then eigenvalues only are required. If <code>job = 'S'</code> , then the Schur form T is required.
<code>compz</code>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <code>compz = 'N'</code> , then no Schur vectors are computed (and the array z is not referenced). If <code>compz = 'I'</code> , then the Schur vectors of H are computed (and the array z is initialized by the routine). If <code>compz = 'V'</code> , then the Schur vectors of A are computed (and the array z must contain the matrix Q on entry).
<code>n</code>	INTEGER. The order of the matrix H ($n \geq 0$).
<code>ilo, ihi</code>	INTEGER. If A has been balanced by <code>?gebal</code> , then <code>ilo</code> and <code>ihi</code> must contain the values returned by <code>?gebal</code> . Otherwise, <code>ilo</code> must be set to 1 and <code>ihi</code> to n .
<code>h, z, work</code>	REAL for <code>shseqr</code>

DOUBLE PRECISION for dhseqr

COMPLEX for chseqr

DOUBLE COMPLEX for zhseqr.

Arrays:

$h(ldh,*)$) The n -by- n upper Hessenberg matrix H .

The second dimension of h must be at least $\max(1, n)$.

$z(ldz,*)$

If $compz = 'V'$, then z must contain the matrix Q from the reduction to Hessenberg form.

If $compz = 'I'$, then z need not be set.

If $compz = 'N'$, then z is not referenced.

The second dimension of z must be

at least $\max(1, n)$ if $compz = 'V'$ or $'I'$;

at least 1 if $compz = 'N'$.

$work(lwork)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, n)$.

ldh INTEGER. The leading dimension of h ; at least $\max(1, n)$.

ldz INTEGER. The leading dimension of z ;

If $compz = 'N'$, then $ldz \geq 1$.

If $compz = 'V'$ or $'I'$, then $ldz \geq \max(1, n)$.

$lwork$ INTEGER. The dimension of the array $work$.

$lwork \geq \max(1, n)$ is sufficient and delivers very good and sometimes optimal performance. However, $lwork$ as large as $11*n$ may be required for optimal performance. A workspace query is recommended to determine the optimal workspace size.

If $lwork = -1$, then a workspace query is assumed; the routine only estimates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w COMPLEX for chseqr

DOUBLE COMPLEX for zhseqr.

Array, size at least $\max(1, n)$. Contains the computed eigenvalues, unless $info > 0$. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).

wr, wi REAL for shseqr

DOUBLE PRECISION for dhseqr

Arrays, size at least $\max(1, n)$ each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless $info > 0$. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).

<i>h</i>	<p>If $info = 0$ and $job = 'S'$, h contains the upper quasi-triangular matrix T from the Schur decomposition (the Schur form).</p> <p>If $info = 0$ and $job = 'E'$, the contents of h are unspecified on exit. (The output value of h when $info > 0$ is given under the description of $info$ below.)</p>
<i>z</i>	<p>If $compz = 'V'$ and $info = 0$, then z contains Q^*Z.</p> <p>If $compz = 'I'$ and $info = 0$, then z contains the unitary or orthogonal matrix Z of the Schur vectors of H.</p> <p>If $compz = 'N'$, then z is not referenced.</p>
<i>work(1)</i>	On exit, if $info = 0$, then <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, <code>?hseqr</code> failed to compute all of the eigenvalues. Elements 1,2, ..., $ilo-1$ and $i+1, i+2, \dots, n$ of the eigenvalue arrays (wr and wi for real flavors and w for complex flavors) contain the real and imaginary parts of those eigenvalues that have been successfully found.</p> <p>If $info > 0$, and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of H.</p> <p>If $info > 0$, and $job = 'S'$, then on exit (initial value of H)*$U = U$*(final value of H), where U is a unitary matrix. The final value of H is upper Hessenberg and triangular in rows and columns $info+1$ through ihi.</p> <p>If $info > 0$, and $compz = 'V'$, then on exit (final value of Z) = (initial value of Z)*U, where U is the unitary matrix (regardless of the value of job).</p> <p>If $info > 0$, and $compz = 'I'$, then on exit (final value of Z) = U, where U is the unitary matrix (regardless of the value of job).</p> <p>If $info > 0$, and $compz = 'N'$, then Z is not accessed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hseqr` interface are the following:

<i>h</i>	Holds the matrix H of size (n,n) .
<i>wr</i>	Holds the vector of length n . Used in real flavors only.

<i>wi</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>w</i>	Holds the vector of length <i>n</i> . Used in complex flavors only.
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n,n</i>).
<i>job</i>	Must be 'E' or 'S'. The default value is 'E'.
<i>compz</i>	<p>If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted.</p> <p>If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.</p>

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\epsilon) \|H\|_2 / s_i$, and ϵ is the machine precision.

If λ_j is an exact eigenvalue, and μ_j is the corresponding computed value, then $|\lambda_j - \mu_j| \leq c(n) * \epsilon * \|H\|_2 / s_i$, where $c(n)$ is a modestly increasing function of n , and s_i is the reciprocal condition number of λ_j . The condition numbers s_i may be computed by calling [trсна](#).

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:	<p>$7n^3$ for real flavors</p> <p>$25n^3$ for complex flavors.</p>
If the Schur form is computed:	<p>$10n^3$ for real flavors</p> <p>$35n^3$ for complex flavors.</p>
If the full Schur factorization is computed:	<p>$20n^3$ for real flavors</p> <p>$70n^3$ for complex flavors.</p>

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

```
call shsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm,
m, work, ifaill, ifailr, info)
```

```

call dhsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm,
m, work, ifaill, ifailr, info)

call chsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m,
work, rwork, ifaill, ifailr, info)

call zhsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m,
work, rwork, ifaill, ifailr, info)

call hsein(h, wr, wi, select [, vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m]
[,info])

call hsein(h, w, select [,vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m] [,info])

```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes left and/or right eigenvectors of an upper Hessenberg matrix H , corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y , corresponding to an eigenvalue λ , are defined by: $H^*x = \lambda^*x$ and $y^{H*}H = \lambda^*y^H$ (or $H^H*y = \lambda^*y$). Here λ^* denotes the conjugate of λ .

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x , $\max |x_i| = 1$, and for a complex eigenvector, $\max(|\text{Re}x_i| + |\text{Im}x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by [ormhr](#) or [unmhr](#).

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'R' or 'L' or 'B'. If <i>side</i> = 'R', then only right eigenvectors are computed. If <i>side</i> = 'L', then only left eigenvectors are computed. If <i>side</i> = 'B', then all eigenvectors are computed.
<i>eigsrc</i>	CHARACTER*1. Must be 'Q' or 'N'. If <i>eigsrc</i> = 'Q', then the eigenvalues of H were found using hseqr ; thus if H has any zero sub-diagonal elements (and so is block triangular), then the j -th eigenvalue can be assumed to be an eigenvalue of the block containing the j -th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If <i>eigsrc</i> = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.
<i>initv</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>initv</i> = 'N', then no initial estimates for the selected eigenvectors are supplied. If <i>initv</i> = 'U', then initial estimates for the selected eigenvectors are supplied in <i>vl</i> and/or <i>vr</i> .
<i>select</i>	LOGICAL.

Array, size at least $\max(1, n)$. Specifies which eigenvectors are to be computed.

For real flavors:

To obtain the real eigenvector corresponding to the real eigenvalue $wr(j)$, set $select(j)$ to `.TRUE.`

To select the complex eigenvector corresponding to the complex eigenvalue $(wr(j), wi(j))$ with complex conjugate $(wr(j+1), wi(j+1))$, set $select(j)$ and/or $select(j+1)$ to `.TRUE.`; the eigenvector corresponding to the first eigenvalue in the pair is computed.

For complex flavors:

To select the eigenvector corresponding to the eigenvalue $w(j)$, set $select(j)$ to `.TRUE.`

n

INTEGER. The order of the matrix H ($n \geq 0$).

$h, vl, vr, work$

REAL for `shsein`

DOUBLE PRECISION for `dhsein`

COMPLEX for `chsein`

DOUBLE COMPLEX for `zhsein`.

Arrays:

$h(ldh,*)$ The n -by- n upper Hessenberg matrix H . If an NAN value is detected in h , the routine returns with $info = -6$.

The second dimension of h must be at least $\max(1, n)$.

$vl(ldvl,*)$

If $initv = 'V'$ and $side = 'L'$ or $'B'$, then vl must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If $initv = 'N'$, then vl need not be set.

The second dimension of vl must be at least $\max(1, mm)$ if $side = 'L'$ or $'B'$ and at least 1 if $side = 'R'$.

The array vl is not referenced if $side = 'R'$.

$vr(ldvr,*)$

If $initv = 'V'$ and $side = 'R'$ or $'B'$, then vr must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If $initv = 'N'$, then vr need not be set.

The second dimension of vr must be at least $\max(1, mm)$ if $side = 'R'$ or $'B'$ and at least 1 if $side = 'L'$.

The array vr is not referenced if $side = 'L'$.

$work(*)$ is a workspace array.

size at least $\max(1, n*(n+2))$ for real flavors and at least $\max(1, n*n)$ for complex flavors.

<i>ldh</i>	INTEGER. The leading dimension of <i>h</i> ; at least $\max(1, n)$.
<i>w</i>	COMPLEX for <i>chsein</i> DOUBLE COMPLEX for <i>zhsein</i> . Array, size at least $\max(1, n)$. Contains the eigenvalues of the matrix <i>H</i> . If <i>eigsrc</i> = 'Q', the array must be exactly as returned by ?hseqr.
<i>wr, wi</i>	REAL for <i>shsein</i> DOUBLE PRECISION for <i>dhsein</i> Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix <i>H</i> . Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If <i>eigsrc</i> = 'Q', the arrays must be exactly as returned by ?hseqr.
<i>ldvl</i>	INTEGER. The leading dimension of <i>vl</i> . If <i>side</i> = 'L' or 'B', $ldvl \geq \max(1, n)$. If <i>side</i> = 'R', $ldvl \geq 1$.
<i>ldvr</i>	INTEGER. The leading dimension of <i>vr</i> . If <i>side</i> = 'R' or 'B', $ldvr \geq \max(1, n)$. If <i>side</i> = 'L', $ldvr \geq 1$.
<i>mm</i>	INTEGER. The number of columns in <i>vl</i> and/or <i>vr</i> . Must be at least <i>m</i> , the actual number of columns required (see <i>Output Parameters</i> below). <i>For real flavors</i> , <i>m</i> is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see <i>select</i>). <i>For complex flavors</i> , <i>m</i> is the number of selected eigenvectors (see <i>select</i>). Constraint: $0 \leq mm \leq n$.
<i>rwork</i>	REAL for <i>chsein</i> DOUBLE PRECISION for <i>zhsein</i> . Array, size at least $\max(1, n)$.

Output Parameters

<i>select</i>	Overwritten for real flavors only. If a complex eigenvector was selected as specified above, then <i>select(j)</i> is set to <code>.TRUE.</code> and <i>select(j + 1)</i> to <code>.FALSE.</code>
<i>w</i>	The real parts of some elements of <i>w</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

<i>wr</i>	Some elements of <i>wr</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.
<i>vl, vr</i>	<p>If <i>side</i> = 'L' or 'B', <i>vl</i> contains the computed left eigenvectors (as specified by <i>select</i>).</p> <p>If <i>side</i> = 'R' or 'B', <i>vr</i> contains the computed right eigenvectors (as specified by <i>select</i>).</p> <p>The eigenvectors treated column-wise form a rectangular <i>n</i>-by-<i>mm</i> matrix.</p> <p><i>For real flavors</i>: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by <i>matrix_layout</i> (using either column major or row major layout).</p>
<i>m</i>	<p>INTEGER. <i>For real flavors</i>: the number of columns of <i>vl</i> and/or <i>vr</i> required to store the selected eigenvectors.</p> <p><i>For complex flavors</i>: the number of selected eigenvectors.</p>
<i>ifaill, ifailr</i>	<p>INTEGER.</p> <p>Arrays, size at least $\max(1, mm)$ each.</p> <p><i>ifaill</i>(<i>i</i>) = 0 if the <i>i</i>th column of <i>vl</i> converged;</p> <p><i>ifaill</i>(<i>i</i>) = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vl</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge.</p> <p><i>ifailr</i>(<i>i</i>) = 0 if the <i>i</i>th column of <i>vr</i> converged;</p> <p><i>ifailr</i>(<i>i</i>) = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vr</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge.</p> <p><i>For real flavors</i>: if the <i>i</i>th and (<i>i</i>+1)th columns of <i>vl</i> contain a selected complex eigenvector, then <i>ifaill</i>(<i>i</i>) and <i>ifaill</i>(<i>i</i> + 1) are set to the same value. A similar rule holds for <i>vr</i> and <i>ifailr</i>.</p> <p>The array <i>ifaill</i> is not referenced if <i>side</i> = 'R'. The array <i>ifailr</i> is not referenced if <i>side</i> = 'L'.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> > 0, then <i>i</i> eigenvectors (as indicated by the parameters <i>ifaill</i> and/or <i>ifailr</i> above) failed to converge. The corresponding columns of <i>vl</i> and/or <i>vr</i> contain no useful information.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hsein` interface are the following:

<i>h</i>	Holds the matrix <i>H</i> of size (n,n) .
<i>wr</i>	Holds the vector of length n . Used in real flavors only.
<i>wi</i>	Holds the vector of length n . Used in real flavors only.
<i>w</i>	Holds the vector of length n . Used in complex flavors only.
<i>select</i>	Holds the vector of length n .
<i>vl</i>	Holds the matrix <i>VL</i> of size (n,mm) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n,mm) .
<i>ifail1</i>	Holds the vector of length (mm) . Note that there will be an error condition if <i>ifail1</i> is present and <i>vl</i> is omitted.
<i>ifailr</i>	Holds the vector of length (mm) . Note that there will be an error condition if <i>ifailr</i> is present and <i>vr</i> is omitted.
<i>initv</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>eigsrc</i>	Must be 'N' or 'Q'. The default value is 'N'.
<i>side</i>	Restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: <i>side</i> = 'B', if both <i>vl</i> and <i>vr</i> are present, <i>side</i> = 'L', if <i>vl</i> is present and <i>vr</i> omitted, <i>side</i> = 'R', if <i>vl</i> is omitted and <i>vr</i> present, Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.

Application Notes

Each computed right eigenvector x_i is the exact eigenvector of a nearby matrix $A + E_i$, such that $\|E_i\| < O(\epsilon) \|A\|$. Hence the residual is small:

$$\|Ax_i - \lambda_i x_i\| = O(\epsilon) \|A\|.$$

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

```
call strevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
call dtrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call ztrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call trevc(t [, howmny] [, select] [, vl] [, vr] [, m] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^* T^* Q^H$, as computed by [hseqr](#).

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w , are defined by:

$$T^* x = w^* x, y^H T = w^* y^H, \text{ where } y^H \text{ denotes the conjugate transpose of } y.$$

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T .

This routine returns the matrices X and/or Y of right and left eigenvectors of T , or the products $Q^* X$ and/or $Q^* Y$, where Q is an input matrix.

If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T , then $Q^* X$ and $Q^* Y$ are the matrices of right and left eigenvectors of A .

Input Parameters

<i>side</i>	<p>CHARACTER*1. Must be 'R' or 'L' or 'B'.</p> <p>If <i>side</i> = 'R', then only right eigenvectors are computed.</p> <p>If <i>side</i> = 'L', then only left eigenvectors are computed.</p> <p>If <i>side</i> = 'B', then all eigenvectors are computed.</p>
<i>howmny</i>	<p>CHARACTER*1. Must be 'A' or 'B' or 'S'.</p> <p>If <i>howmny</i> = 'A', then all eigenvectors (as specified by <i>side</i>) are computed.</p> <p>If <i>howmny</i> = 'B', then all eigenvectors (as specified by <i>side</i>) are computed and backtransformed by the matrices supplied in <i>vl</i> and <i>vr</i>.</p> <p>If <i>howmny</i> = 'S', then selected eigenvectors (as specified by <i>side</i> and <i>select</i>) are computed.</p>
<i>select</i>	<p>LOGICAL.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>howmny</i> = 'S', <i>select</i> specifies which eigenvectors are to be computed.</p> <p>If <i>howmny</i> = 'A' or 'B', <i>select</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>If <i>omega(j)</i> is a real eigenvalue, the corresponding real eigenvector is computed if <i>select(j)</i> is <code>.TRUE.</code></p> <p>If <i>omega(j)</i> and <i>omega(j + 1)</i> are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either <i>select(j)</i> or <i>select(j + 1)</i> is <code>.TRUE.</code>, and on exit <i>select(j)</i> is set to <code>.TRUE.</code> and <i>select(j + 1)</i> is set to <code>.FALSE.</code></p> <p><i>For complex flavors:</i></p> <p>The eigenvector corresponding to the <i>j</i>-th eigenvalue is computed if <i>select(j)</i> is <code>.TRUE.</code></p>

n INTEGER. The order of the matrix T ($n \geq 0$).

t, vl, vr REAL for `strevc`
 DOUBLE PRECISION for `dtrevc`
 COMPLEX for `ctrevc`
 DOUBLE COMPLEX for `ztrevc`.

Arrays:

t(ldt,)* contains the n -by- n matrix T in Schur canonical form. For complex flavors `ctrevc` and `ztrevc`, contains the upper triangular matrix T .

The second dimension of *t* must be at least $\max(1, n)$.

vl(ldvl,)*

If *howmny* = 'B' and *side* = 'L' or 'B', then *vl* must contain an n -by- n matrix Q (usually the matrix of Schur vectors returned by `?hseqr`).

If *howmny* = 'A' or 'S', then *vl* need not be set.

The second dimension of *vl* must be at least $\max(1, mm)$ if *side* = 'L' or 'B' and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

vr(ldvr,)*

If *howmny* = 'B' and *side* = 'R' or 'B', then *vr* must contain an n -by- n matrix Q (usually the matrix of Schur vectors returned by `?hseqr`). .

If *howmny* = 'A' or 'S', then *vr* need not be set.

The second dimension of *vr* must be at least $\max(1, mm)$ if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.

work()* is a workspace array.

size at least $\max(1, 3*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

ldt INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

ldvl INTEGER. The leading dimension of *vl*.

If *side* = 'L' or 'B', $ldvl \geq n$.

If *side* = 'R', $ldvl \geq 1$.

ldvr INTEGER. The leading dimension of *vr*.

If *side* = 'R' or 'B', $ldvr \geq n$.

If *side* = 'L', $ldvr \geq 1$.

mm INTEGER. The number of columns in the arrays *vl* and/or *vr*. Must be at least m (the precise number of columns required).

If *howmny* = 'A' or 'B', $mm = n$.

If *howmny* = 'S': for real flavors, *mm* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector;

for complex flavors, mm is the number of selected eigenvectors (see *select*).

Constraint: $0 \leq mm \leq n$.

rwork

REAL for *ctrevc*

DOUBLE PRECISION for *ztrevc*.

Workspace array, size at least $\max(1, n)$.

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then *select(j)* is set to `.TRUE.` and *select(j + 1)* to `.FALSE.`

t

COMPLEX for *ctrevc*

DOUBLE COMPLEX for *ztrevc*.

ctrevc/ztrevc modify the *t(ldt,*)* array, which is restored on exit.

vl, vr

If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).

If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).

The eigenvectors treated column-wise form a rectangular n -by- mm matrix.

For real flavors: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by *matrix_layout* (using either column major or row major layout).

m

INTEGER.

For complex flavors: the number of selected eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to n .

For real flavors: the number of columns of *vl* and/or *vr* actually used to store the selected eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to n .

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *trevc* interface are the following:

<i>t</i>	Holds the matrix <i>T</i> of size (n,n) .
<i>select</i>	Holds the vector of length n .
<i>vl</i>	Holds the matrix <i>VL</i> of size (n,mm) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n,mm) .
<i>side</i>	If omitted, this argument is restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: <i>side</i> = 'B', if both <i>vl</i> and <i>vr</i> are present, <i>side</i> = 'L', if <i>vr</i> is omitted, <i>side</i> = 'R', if <i>vl</i> is omitted. Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.
<i>howmny</i>	If omitted, this argument is restored based on the presence of argument <i>select</i> as follows: <i>howmny</i> = 'V', if <i>q</i> is present, <i>howmny</i> = 'N', if <i>q</i> is omitted. If present, <i>vect</i> = 'V' or 'U' and the argument <i>q</i> must also be present. Note that there will be an error condition if both <i>select</i> and <i>howmny</i> are present.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\epsilon \|T\|_2) / \text{sep}_i$ where sep_i is the reciprocal condition number of x_i . The condition number sep_i may be computed by calling `?trsna`.

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

```
call strsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, iwork, info)
call dtrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, iwork, info)
call ctrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, rwork, info)
call ztrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, rwork, info)
call trsna(t [, s] [, sep] [, vl] [, vr] [, select] [, m] [, info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z^* T^* Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue λ_i as $s_i = |v^T u| / (|u|_E |v|_E)$ for real flavors and $s_i = |v^H u| / (|u|_E |v|_E)$ for complex flavors,

where:

- u and v are the right and left eigenvectors of T , respectively, corresponding to λ_i .
- v^T/v^H denote transpose/conjugate transpose of v , respectively.

This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue λ_i is then given by $\epsilon^* ||T|| / s_i$, where ϵ is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to λ_i , the routine first calls `trexc` to reorder the diagonal elements of matrix T so that λ_i is in the leading position:

```
.....
.....
```

The reciprocal condition number of the eigenvector is then estimated as sep_i , the smallest singular value of the matrix $T_{22} - \lambda_i I$.

An approximate error estimate for a computed right eigenvector u corresponding to λ_i is then given by $\epsilon^* ||T|| / sep_i$.

Input Parameters

<code>job</code>	<p>CHARACTER*1. Must be 'E' or 'V' or 'B'.</p> <p>If <code>job = 'E'</code>, then condition numbers for eigenvalues only are computed.</p> <p>If <code>job = 'V'</code>, then condition numbers for eigenvectors only are computed.</p> <p>If <code>job = 'B'</code>, then condition numbers for both eigenvalues and eigenvectors are computed.</p>
<code>howmny</code>	<p>CHARACTER*1. Must be 'A' or 'S'.</p> <p>If <code>howmny = 'A'</code>, then the condition numbers for all eigenpairs are computed.</p> <p>If <code>howmny = 'S'</code>, then condition numbers for selected eigenpairs (as specified by <code>select</code>) are computed.</p>
<code>select</code>	<p>LOGICAL.</p> <p>Array, size at least $\max(1, n)$ if <code>howmny = 'S'</code> and at least 1 otherwise.</p> <p>Specifies the eigenpairs for which condition numbers are to be computed if <code>howmny = 'S'</code>.</p> <p><i>For real flavors:</i></p> <p>To select condition numbers for the eigenpair corresponding to the real eigenvalue λ_j, <code>select(j)</code> must be set <code>.TRUE.</code>;</p>

to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues λ_j and λ_{j+1} , *select(j)* and/or *select(j + 1)* must be set `.TRUE.`

For complex flavors

To select condition numbers for the eigenpair corresponding to the eigenvalue λ_j , *select(j)* must be set `.TRUE.` *select* is not referenced if *howmny* = 'A'.

n INTEGER. The order of the matrix *T* ($n \geq 0$).

t, vl, vr, work REAL for *strsna*
DOUBLE PRECISION for *dtrsna*
COMPLEX for *ctrsna*
DOUBLE COMPLEX for *ztrsna*.

Arrays:

t(ldt,)* contains the *n*-by-*n* matrix *T*.
The second dimension of *t* must be at least $\max(1, n)$.

vl(ldvl,)*
If *job* = 'E' or 'B', then *vl* must contain the left eigenvectors of *T* (or of any matrix $Q^*T^*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by [trevc](#) or [hsein](#).
The second dimension of *vl* must be at least $\max(1, mm)$ if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.
The array *vl* is not referenced if *job* = 'V'.

vr(ldvr,)*
If *job* = 'E' or 'B', then *vr* must contain the right eigenvectors of *T* (or of any matrix $Q^*T^*Q^H$ with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by [trevc](#) or [hsein](#).
The second dimension of *vr* must be at least $\max(1, mm)$ if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.
The array *vr* is not referenced if *job* = 'V'.

work is a workspace array, its dimension (*ldwork*, *n*+6).
The array *work* is not referenced if *job* = 'E'.

ldt INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

ldvl INTEGER. The leading dimension of *vl*.
If *job* = 'E' or 'B', $ldvl \geq \max(1, n)$.
If *job* = 'V', $ldvl \geq 1$.

ldvr INTEGER. The leading dimension of *vr*.
If *job* = 'E' or 'B', $ldvr \geq \max(1, n)$.

	If $job = 'R'$, $ldvr \geq 1$.
<i>mm</i>	<p>INTEGER. The number of elements in the arrays <i>s</i> and <i>sep</i>, and the number of columns in <i>vl</i> and <i>vr</i> (if used). Must be at least <i>m</i> (the precise number required).</p> <p>If $howmny = 'A'$, $mm = n$;</p> <p>if $howmny = 'S'$, for real flavors <i>mm</i> is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.</p> <p>for complex flavors <i>mm</i> is the number of selected eigenpairs (see <i>select</i>).</p> <p>Constraint:</p> <p>$0 \leq mm \leq n$.</p>
<i>ldwork</i>	<p>INTEGER. The leading dimension of <i>work</i>.</p> <p>If $job = 'V'$ or $'B'$, $ldwork \geq \max(1, n)$.</p> <p>If $job = 'E'$, $ldwork \geq 1$.</p>
<i>rwork</i>	<p>REAL for <i>ctrсна</i>, <i>ztrsна</i>.</p> <p>Array, size at least $\max(1, n)$. The array is not referenced if $job = 'E'$.</p>
<i>iwork</i>	<p>INTEGER for <i>strсна</i>, <i>dtrsна</i>.</p> <p>Array, size at least $\max(1, 2*(n - 1))$. The array is not referenced if $job = 'E'$.</p>

Output Parameters

<i>s</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array, size at least $\max(1, mm)$ if $job = 'E'$ or $'B'$ and at least 1 if $job = 'V'$.</p> <p>Contains the reciprocal condition numbers of the selected eigenvalues if $job = 'E'$ or $'B'$, stored in consecutive elements of the array. Thus $s(j)$, $sep(j)$ and the <i>j</i>-th columns of <i>vl</i> and <i>vr</i> all correspond to the same eigenpair (but not in general the <i>j</i> th eigenpair unless all eigenpairs have been selected).</p> <p>For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of <i>s</i> are set to the same value. The array <i>s</i> is not referenced if $job = 'V'$.</p>
<i>sep</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array, size at least $\max(1, mm)$ if $job = 'V'$ or $'B'$ and at least 1 if $job = 'E'$. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if $job = 'V'$ or $'B'$, stored in consecutive elements of the array.</p>

For real flavors: for a complex eigenvector, two consecutive elements of *sep* are set to the same value; if the eigenvalues cannot be reordered to compute *sep(j)*, then *sep(j)* is set to zero; this can only occur when the true value would be very small anyway. The array *sep* is not referenced if *job* = 'E'.

m INTEGER.

For complex flavors: the number of selected eigenpairs.

If *howmny* = 'A', *m* is set to *n*.

For real flavors: the number of elements of *s* and/or *sep* actually used to store the estimated condition numbers.

If *howmny* = 'A', *m* is set to *n*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trsna` interface are the following:

<i>t</i>	Holds the matrix <i>T</i> of size (<i>n,n</i>).
<i>s</i>	Holds the vector of length (<i>mm</i>).
<i>sep</i>	Holds the vector of length (<i>mm</i>).
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n,mm</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n,mm</i>).
<i>select</i>	Holds the vector of length <i>n</i> .
<i>job</i>	Restored based on the presence of arguments <i>s</i> and <i>sep</i> as follows: <i>job</i> = 'B', if both <i>s</i> and <i>sep</i> are present, <i>job</i> = 'E', if <i>s</i> is present and <i>sep</i> omitted, <i>job</i> = 'V', if <i>s</i> is omitted and <i>sep</i> present. Note an error condition if both <i>s</i> and <i>sep</i> are omitted.
<i>howmny</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>howmny</i> = 'S', if <i>select</i> is present, <i>howmny</i> = 'A', if <i>select</i> is omitted.

Note that the arguments *s*, *vl*, and *vr* must either be all present or all omitted.

Otherwise, an error condition is observed.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc

Reorders the Schur factorization of a general matrix.

Syntax

```
call strexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call dtrexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call ctrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call ztrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call trexc(t, ifst, ilst [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reorders the Schur factorization of a general matrix $A = Q^*T^*Q^H$, so that the diagonal element or block of T with row index $ifst$ is moved to row $ilst$.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H * T * Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q * Z$, giving $A = P * S * P^H$.

Input Parameters

<i>compq</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>compq</i> = 'V', then the Schur vectors (Q) are updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>t, q</i>	REAL for strexc DOUBLE PRECISION for dtrexc COMPLEX for ctrexc DOUBLE COMPLEX for ztrexc. Arrays: <i>t(ldt,*)</i> contains the n -by- n matrix T . The second dimension of <i>t</i> must be at least $\max(1, n)$. <i>q(ldq,*)</i> If <i>compq</i> = 'V', then <i>q</i> must contain Q (Schur vectors). If <i>compq</i> = 'N', then <i>q</i> is not referenced. The second dimension of <i>q</i> must be at least $\max(1, n)$ if <i>compq</i> = 'V' and at least 1 if <i>compq</i> = 'N'.

<i>ldt</i>	INTEGER. The leading dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldq</i>	INTEGER. The leading dimension of <i>q</i> ; If <i>compq</i> = 'N', then $ldq \geq 1$. If <i>compq</i> = 'V', then $ldq \geq \max(1, n)$.
<i>ifst, ilst</i>	INTEGER. $1 \leq ifst \leq n$; $1 \leq ilst \leq n$. Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix <i>T</i> . The element (or block) with row index <i>ifst</i> is moved to row <i>ilst</i> by a sequence of exchanges between adjacent elements (or blocks).
<i>work</i>	REAL for <i>strex</i> DOUBLE PRECISION for <i>dtrex</i> . Array, size at least $\max(1, n)$.

Output Parameters

<i>t</i>	Overwritten by the updated matrix <i>S</i> .
<i>q</i>	If <i>compq</i> = 'V', <i>q</i> contains the updated matrix of Schur vectors.
<i>ifst, ilst</i>	Overwritten for real flavors only. If <i>ifst</i> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <i>ilst</i> always points to the first row of the block in its final position (which may differ from its input value by ± 1).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *trexc* interface are the following:

<i>t</i>	Holds the matrix <i>T</i> of size (n, n) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>compq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>compq</i> = 'V', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted.

Application Notes

The computed matrix *S* is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

<i>job</i>	<p>CHARACTER*1. Must be 'N' or 'E' or 'V' or 'B'.</p> <p>If <i>job</i> = 'N', then no condition numbers are required.</p> <p>If <i>job</i> = 'E', then only the condition number for the cluster of eigenvalues is computed.</p> <p>If <i>job</i> = 'V', then only the condition number for the invariant subspace is computed.</p> <p>If <i>job</i> = 'B', then condition numbers for both the cluster and the invariant subspace are computed.</p>
<i>compq</i>	<p>CHARACTER*1. Must be 'V' or 'N'.</p> <p>If <i>compq</i> = 'V', then <i>Q</i> of the Schur vectors is updated.</p> <p>If <i>compq</i> = 'N', then no Schur vectors are updated.</p>
<i>select</i>	<p>LOGICAL.</p> <p>Array, size at least $\max(1, n)$.</p> <p>Specifies the eigenvalues in the selected cluster. To select an eigenvalue λ_j, <i>select(j)</i> must be <code>.TRUE.</code></p> <p><i>For real flavors:</i> to select a complex conjugate pair of eigenvalues λ_j and λ_{j+1} (corresponding 2 by 2 diagonal block), <i>select(j)</i> and/or <i>select(j + 1)</i> must be <code>.TRUE.</code>; the complex conjugate λ_j and λ_{j+1} must be either both included in the cluster or both excluded.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>T</i> ($n \geq 0$).</p>
<i>t, q, work</i>	<p>REAL for <i>strsen</i></p> <p>DOUBLE PRECISION for <i>dtrsen</i></p> <p>COMPLEX for <i>ctrsen</i></p> <p>DOUBLE COMPLEX for <i>ztrsen</i>.</p> <p>Arrays:</p> <p><i>t(ldt,*)</i> The upper quasi-triangular <i>n</i>-by-<i>n</i> matrix <i>T</i>, in Schur canonical form. The second dimension of <i>t</i> must be at least $\max(1, n)$.</p> <p><i>q(ldq,*)</i></p> <p>If <i>compq</i> = 'V', then <i>q</i> must contain the matrix <i>Q</i> of Schur vectors.</p> <p>If <i>compq</i> = 'N', then <i>q</i> is not referenced.</p> <p>The second dimension of <i>q</i> must be at least $\max(1, n)$ if <i>compq</i> = 'V' and at least 1 if <i>compq</i> = 'N'.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of <i>t</i>; at least $\max(1, n)$.</p>
<i>ldq</i>	<p>INTEGER. The leading dimension of <i>q</i>;</p>

If $compq = 'N'$, then $ldq \geq 1$.

If $compq = 'V'$, then $ldq \geq \max(1, n)$.

lwork

INTEGER. The dimension of the array *work*.

If $job = 'V'$ or $'B'$, $lwork \geq \max(1, 2*m*(n-m))$.

If $job = 'E'$, then $lwork \geq \max(1, m*(n-m))$

If $job = 'N'$, then $lwork \geq 1$ for complex flavors and $lwork \geq \max(1, n)$ for real flavors.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.*iwork*(*liwork*) is a workspace array. The array *iwork* is not referenced if $job = 'N'$ or $'E'$.

The actual amount of workspace required cannot exceed $n^2/2$ if $job = 'V'$ or $'B'$.

liwork

INTEGER.

The dimension of the array *iwork*.

If $job = 'V'$ or $'B'$, $liwork \geq \max(1, 2m(n-m))$.

If $job = 'E'$ or $'E'$, $liwork \geq 1$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

t

Overwritten by the reordered matrix *R* in Schur canonical form with the selected eigenvalues in the leading diagonal blocks.

q

If $compq = 'V'$, *q* contains the updated matrix of Schur vectors; the first *m* columns of the *Q* form an orthogonal basis for the specified invariant subspace.

w

COMPLEX for *ctrsen*

DOUBLE COMPLEX for *ztrsen*.

Array, size at least $\max(1, n)$. The recorded eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*.

wr, wi

REAL for *strsen*

DOUBLE PRECISION for *dtrsen*

Arrays, size at least $\max(1, n)$. Contain the real and imaginary parts, respectively, of the reordered eigenvalues of *R*. The eigenvalues are stored in the same order as on the diagonal of *R*. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

<i>m</i>	<p>INTEGER.</p> <p><i>For complex flavors:</i> the dimension of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see <i>select</i>).</p> <p><i>For real flavors:</i> the dimension of the specified invariant subspace. The value of <i>m</i> is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see <i>select</i>).</p> <p>Constraint: $0 \leq m \leq n$.</p>
<i>s</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>If <i>job</i> = 'E' or 'B', <i>s</i> is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.</p> <p>If <i>m</i> = 0 or <i>n</i>, then <i>s</i> = 1.</p> <p><i>For real flavors:</i> if <i>info</i> = 1, then <i>s</i> is set to zero. <i>s</i> is not referenced if <i>job</i> = 'N' or 'V'.</p>
<i>sep</i>	<p>REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.</p> <p>If <i>job</i> = 'V' or 'B', <i>sep</i> is the estimated reciprocal condition number of the specified invariant subspace.</p> <p>If <i>m</i> = 0 or <i>n</i>, then <i>sep</i> = <i>T</i> .</p> <p><i>For real flavors:</i> if <i>info</i> = 1, then <i>sep</i> is set to zero.</p> <p><i>sep</i> is not referenced if <i>job</i> = 'N' or 'E'.</p>
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the optimal size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the optimal size of <i>liwork</i> .
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = 1, the reordering of <i>T</i> failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); <i>T</i> may have been partially reordered, and <i>wr</i> and <i>wi</i> contain the eigenvalues in the same order as in <i>T</i>; <i>s</i> and <i>sep</i> (if requested) are set to zero.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trsen` interface are the following:

<i>t</i>	Holds the matrix <i>T</i> of size (<i>n</i> , <i>n</i>).
<i>select</i>	Holds the vector of length <i>n</i> .
<i>wr</i>	Holds the vector of length <i>n</i> . Used in real flavors only.

<i>wi</i>	Holds the vector of length n . Used in real flavors only.
<i>w</i>	Holds the vector of length n . Used in complex flavors only.
<i>q</i>	Holds the matrix Q of size (n,n) .
<i>compq</i>	Restored based on the presence of the argument q as follows: $compq = 'V'$, if q is present, $compq = 'N'$, if q is omitted.
<i>job</i>	Restored based on the presence of arguments s and sep as follows: $job = 'B'$, if both s and sep are present, $job = 'E'$, if s is present and sep omitted, $job = 'V'$, if s is omitted and sep present, $job = 'N'$, if both s and sep are omitted.

Application Notes

The computed matrix R is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision. The computed s cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))_{1/2}$; sep may differ from the true value by $(m*n-m^2)_{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\epsilon) * \|A\|_2 / sep$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real.

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run or set $lwork = -1$ ($liwork = -1$).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

```
call strsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call dtrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ctrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ztrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call trsyl(a, b, c, scale [, trana] [,tranb] [,isgn] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine solves the Sylvester matrix equation $\text{op}(A) * X \pm X * \text{op}(B) = \alpha * C$, where $\text{op}(A) = A$ or A^H , and the matrices A and B are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in X ; A is m -by- m , B is n -by- n , and C and X are both m -by- n . The matrix X is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B , respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

<i>trana</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trana</i> = 'N', then $\text{op}(A) = A$. If <i>trana</i> = 'T', then $\text{op}(A) = A^T$ (real flavors only). If <i>trana</i> = 'C' then $\text{op}(A) = A^H$.
<i>tranb</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>tranb</i> = 'N', then $\text{op}(B) = B$. If <i>tranb</i> = 'T', then $\text{op}(B) = B^T$ (real flavors only). If <i>tranb</i> = 'C', then $\text{op}(B) = B^H$.
<i>isgn</i>	INTEGER. Indicates the form of the Sylvester equation. If <i>isgn</i> = +1, $\text{op}(A) * X + X * \text{op}(B) = \text{alpha} * C$. If <i>isgn</i> = -1, $\text{op}(A) * X - X * \text{op}(B) = \text{alpha} * C$.
<i>m</i>	INTEGER. The order of A , and the number of rows in X and C ($m \geq 0$).
<i>n</i>	INTEGER. The order of B , and the number of columns in X and C ($n \geq 0$).
<i>a, b, c</i>	REAL for <code>strsyl</code> DOUBLE PRECISION for <code>dtrsyl</code> COMPLEX for <code>ctrsyl</code> DOUBLE COMPLEX for <code>ztrsyl</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, m)$. <i>b</i> (<i>ldb</i> ,*) contains the matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

ldc INTEGER. The leading dimension of *c*; at least $\max(1, m)$.

Output Parameters

c Overwritten by the solution matrix *X*.

scale REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
The value of the scale factor α .

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = 1, *A* and *B* have common or close eigenvalues; perturbed values were used to solve the equation.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `trsyl` interface are the following:

a Holds the matrix *A* of size (m,m) .

b Holds the matrix *B* of size (n,n) .

c Holds the matrix *C* of size (m,n) .

trana Must be 'N', 'C', or 'T'. The default value is 'N'.

tranb Must be 'N', 'C', or 'T'. The default value is 'N'.

isgn Must be +1 or -1. The default value is +1.

Application Notes

Let *X* be the exact, *Y* the corresponding computed solution, and *R* the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$$\|R\|_F = O(\epsilon) * (\|A\|_F + \|B\|_F) * \|Y\|_F.$$

However, *Y* is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$\|Y - X\|_F \leq \|R\|_F / \text{sep}(A, B)$$

but this may be a considerable overestimate. See [[Golub96](#)] for a definition of $\text{sep}(A, B)$.

The approximate number of floating-point operations for real flavors is $m * n * (m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n -by- n matrices A and B , find the *generalized eigenvalues* λ and the corresponding *generalized eigenvectors* x and y that satisfy the equations

$$Ax = \lambda Bx \text{ (right generalized eigenvectors } x)$$

and

$$y^H A = \lambda y^H B \text{ (left generalized eigenvectors } y).$$

Table "Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems" lists LAPACK routines (FORTRAN 77 interface) used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
ggghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
ggbal	Balances a pair of general real or complex matrices.
ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).
tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
tgseq	Reorders the <i>generalized</i> Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
tgsyl	Solves the generalized Sylvester equation.
zgsyl	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?ggghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

```
call sgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call dgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call cgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call zgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call ggghrd(a, b [,ilo] [,ihi] [,q] [,z] [,compq] [,compz] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reduces a pair of real/complex matrices (A,B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $A^*x = \lambda^*B^*x$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H :

$$Q^H A Z = H$$

and transforms B to another upper triangular matrix T :

$$Q^H B Z = T$$

in order to reduce the problem to its standard form $H^*y = \lambda^*T^*y$, where $y = Z^H x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1 , so that

$$Q_1^* A Z_1^H = (Q_1^* Q)^* H^* (Z_1^* Z)^H$$

$$Q_1^* B Z_1^H = (Q_1^* Q)^* T^* (Z_1^* Z)^H$$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A^*x = \lambda^*B^*x$, then the routine `?gghrd` reduces the original problem to generalized Hessenberg form.

Input Parameters

<i>compq</i>	<p>CHARACTER*1. Must be 'N', 'I', or 'V'.</p> <p>If <i>compq</i> = 'N', matrix Q is not computed.</p> <p>If <i>compq</i> = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;</p> <p>If <i>compq</i> = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1^*Q is returned.</p>
<i>compz</i>	<p>CHARACTER*1. Must be 'N', 'I', or 'V'.</p> <p>If <i>compz</i> = 'N', matrix Z is not computed.</p> <p>If <i>compz</i> = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;</p> <p>If <i>compz</i> = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1^*Z is returned.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>ilo, ihi</i>	<p>INTEGER. <i>ilo</i> and <i>ihi</i> mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$. Values of <i>ilo</i> and <i>ihi</i> are normally set by a previous call to <code>ggbal</code>; otherwise they should be set to 1 and n respectively.</p> <p>Constraint:</p> <p>If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;</p> <p>if $n = 0$, then $ilo = 1$ and $ihi = 0$.</p>

a, b, q, z	<p>REAL for sgghrd</p> <p>DOUBLE PRECISION for dgghrd</p> <p>COMPLEX for cgghrd</p> <p>DOUBLE COMPLEX for zgghrd.</p> <p>Arrays:</p> <p>$a(lda,*)$ contains the n-by-n general matrix A.</p> <p>The second dimension of a must be at least $\max(1, n)$.</p> <p>$b(lb,*)$ contains the n-by-n upper triangular matrix B.</p> <p>The second dimension of b must be at least $\max(1, n)$.</p> <p>$q(ldq,*)$</p> <p>If $compq = 'N'$, then q is not referenced.</p> <p>If $compq = 'V'$, then q must contain the orthogonal/unitary matrix Q_1, typically from the QR factorization of B.</p> <p>The second dimension of q must be at least $\max(1, n)$.</p> <p>$z(ldz,*)$</p> <p>If $compz = 'N'$, then z is not referenced.</p> <p>If $compz = 'V'$, then z must contain the orthogonal/unitary matrix Z_1.</p> <p>The second dimension of z must be at least $\max(1, n)$.</p>
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
ldq	<p>INTEGER. The leading dimension of q;</p> <p>If $compq = 'N'$, then $ldq \geq 1$.</p> <p>If $compq = 'I'$ or $'V'$, then $ldq \geq \max(1, n)$.</p>
ldz	<p>INTEGER. The leading dimension of z;</p> <p>If $compz = 'N'$, then $ldz \geq 1$.</p> <p>If $compz = 'I'$ or $'V'$, then $ldz \geq \max(1, n)$.</p>

Output Parameters

a	On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H , and the rest is set to zero.
b	On exit, overwritten by the upper triangular matrix $T = Q^H * B * Z$. The elements below the diagonal are set to zero.
q	<p>If $compq = 'I'$, then q contains the orthogonal/unitary matrix Q ;</p> <p>If $compq = 'V'$, then q is overwritten by the product $Q_1 * Q$.</p>
z	<p>If $compz = 'I'$, then z contains the orthogonal/unitary matrix Z;</p> <p>If $compz = 'V'$, then z is overwritten by the product $Z_1 * Z$.</p>

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gghrd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>compq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted. If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

```
call sggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call dggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call cggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call zggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call ggbal(a, b [,ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine balances a pair of general real/complex matrices (A, B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to $ilo-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $A^*x = \lambda^*B^*x$.

Input Parameters

job CHARACTER*1. Specifies the operations to be performed on A and B . Must be 'N' or 'P' or 'S' or 'B'.
 If *job* = 'N', then no operations are done; simply set $ilo = 1$, $ihi = n$, $lscale(i) = 1.0$ and $rscale(i) = 1.0$ for $i = 1, \dots, n$.
 If *job* = 'P', then permute only.
 If *job* = 'S', then scale only.
 If *job* = 'B', then both permute and scale.

n INTEGER. The order of the matrices A and B ($n \geq 0$).

a, b REAL for sggbal
 DOUBLE PRECISION for dggbal
 COMPLEX for cggbal
 DOUBLE COMPLEX for zggbal.
 Arrays:
a(*lda*,*) contains the matrix A . The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the matrix B . The second dimension of *b* must be at least $\max(1, n)$.
 If *job* = 'N', *a* and *b* are not referenced.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

work REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Workspace array, size at least $\max(1, 6n)$ when *job* = 'S' or 'B', or at least 1 when *job* = 'N' or 'P'.

Output Parameters

a, b Overwritten by the balanced matrices A and B , respectively.

ilo, ihi INTEGER. *ilo* and *ihi* are set to integers such that on exit $A_{i,j} = 0$ and $B_{i,j} = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.
 If *job* = 'N' or 'S', then $ilo = 1$ and $ihi = n$.

lscale, *rscale* REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Arrays, size at least $\max(1, n)$.
lscale contains details of the permutations and scaling factors applied to the left side of *A* and *B*.
 If P_j is the index of the row interchanged with row j , and D_j is the scaling factor applied to row j , then

$$lscale(j) = P_j, \text{ for } j = 1, \dots, ilo-1$$

$$= D_j, \text{ for } j = ilo, \dots, ihi$$

$$= P_j, \text{ for } j = ihi+1, \dots, n.$$
rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.
 If P_j is the index of the column interchanged with column j , and D_j is the scaling factor applied to column j , then

$$rscale(j) = P_j, \text{ for } j = 1, \dots, ilo-1$$

$$= D_j, \text{ for } j = ilo, \dots, ihi$$

$$= P_j, \text{ for } j = ihi+1, \dots, n$$
 The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ggbal` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n,n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n,n) .
<i>lscale</i>	Holds the vector of length (n) .
<i>rscale</i>	Holds the vector of length (n) .
<i>ilo</i>	Default value for this argument is $ilo = 1$.
<i>ihi</i>	Default value for this argument is $ihi = n$.
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call ggbak(v [, ilo] [, ihi] [, lscale] [, rscale] [, job] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$A*x = \lambda*B*x$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by [ggbal](#).

Input Parameters

<i>job</i>	<p>CHARACTER*1. Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'.</p> <p>If <i>job</i> = 'N', then no operations are done; return.</p> <p>If <i>job</i> = 'P', then do backward transformation for permutation only.</p> <p>If <i>job</i> = 'S', then do backward transformation for scaling only.</p> <p>If <i>job</i> = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument <i>job</i> supplied to ?ggbal.</p>
<i>side</i>	<p>CHARACTER*1. Must be 'L' or 'R'.</p> <p>If <i>side</i> = 'L', then <i>v</i> contains left eigenvectors.</p> <p>If <i>side</i> = 'R', then <i>v</i> contains right eigenvectors.</p>
<i>n</i>	<p>INTEGER. The number of rows of the matrix <i>V</i> ($n \geq 0$).</p>
<i>ilo, ihi</i>	<p>INTEGER. The integers <i>ilo</i> and <i>ihi</i> determined by ?ggbal. Constraint:</p> <p>If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;</p> <p>if $n = 0$, then $ilo = 1$ and $ihi = 0$.</p>
<i>lscale, rscale</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Arrays, size at least $\max(1, n)$.</p> <p>The array <i>lscale</i> contains details of the permutations and/or scaling factors applied to the left side of <i>A</i> and <i>B</i>, as returned by ?ggbal.</p> <p>The array <i>rscale</i> contains details of the permutations and/or scaling factors applied to the right side of <i>A</i> and <i>B</i>, as returned by ?ggbal.</p>
<i>m</i>	<p>INTEGER. The number of columns of the matrix <i>V</i></p>

($m \geq 0$).

v REAL for sggbak
 DOUBLE PRECISION for dggbak
 COMPLEX for cggbak
 DOUBLE COMPLEX for zggbak.
 Array $v(ldv,*)$. Contains the matrix of right or left eigenvectors to be transformed, as returned by [tgevc](#).
 The second dimension of *v* must be at least $\max(1, m)$.

ldv INTEGER. The leading dimension of *v*; at least $\max(1, n)$.

Output Parameters

v Overwritten by the transformed eigenvectors

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *ggbak* interface are the following:

v Holds the matrix *V* of size (*n*,*m*).

lscale Holds the vector of length *n*.

rscale Holds the vector of length *n*.

ilo Default value for this argument is *ilo* = 1.

ihi Default value for this argument is *ihi* = *n*.

job Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

side If omitted, this argument is restored based on the presence of arguments *lscale* and *rscale* as follows:
side = 'L', if *lscale* is present and *rscale* omitted,
side = 'R', if *lscale* is omitted and *rscale* present.
 Note that there will be an error condition if both *lscale* and *rscale* are present or if they both are omitted.

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).

Syntax

```
call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphas, alphai, beta, q,
ldq, z, ldz, work, lwork, info)
```

```
call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphas, alphai, beta, q,
ldq, z, ldz, work, lwork, info)
```

```
call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z,
ldz, work, lwork, rwork, info)
```

```
call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z,
ldz, work, lwork, rwork, info)
```

```
call hgeqz(h, t [,ilo] [,ihi] [,alphas] [,alphai] [,beta] [,q] [,z] [,job] [,compq]
[,compz] [,info])
```

```
call hgeqz(h, t [,ilo] [,ihi] [,alpha] [,beta] [,q] [,z] [,job] [,compq] [,compz]
[,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the eigenvalues of a real/complex matrix pair (H, T) , where H is an upper Hessenberg matrix and T is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (A, B) :

$$A = Q_1^* H^* Z_1^H, B = Q_1^* T^* Z_1^H,$$

as computed by ?gghrd.

For real flavors:

If $job = 'S'$, then the Hessenberg-triangular pair (H, T) is reduced to generalized Schur form,

$$H = Q^* S^* Z^T, T = Q^* P^* Z^T,$$

where Q and Z are orthogonal matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (H, T) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form, that is, if $S_{j+1, j}$ is non-zero, then $P_{j+1, j} = P_j, j+1 = 0, P_j, j > 0$, and $P_{j+1, j+1} > 0$.

For complex flavors:

If $job = 'S'$, then the Hessenberg-triangular pair (H, T) is reduced to generalized Schur form,

$$H = Q^* S^* Z^H, T = Q^* P^* Z^H,$$

where Q and Z are unitary matrices, and S and P are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be post-multiplied by an input matrix Q_1 , and the orthogonal/unitary matrix Z may be post-multiplied by an input matrix Z_1 .

If Q_1 and Z_1 are the orthogonal/unitary matrices from ?gghrd that reduced the matrix pair (A, B) to generalized upper Hessenberg form, then the output matrices $Q_1 Q$ and $Z_1 Z$ are the orthogonal/unitary factors from the generalized Schur factorization of (A, B) :

$$A = (Q_1 Q)^* S^* (Z_1 Z)^H, B = (Q_1 Q)^* P^* (Z_1 Z)^H.$$

To avoid overflow, eigenvalues of the matrix pair (H,T) (equivalently, of (A,B)) are computed as a pair of values (α, β) . For `chgeqz/zhgeqz`, α and β are complex, and for `shgeqz/dhgeqz`, α is complex and β real. If β is nonzero, $\lambda = \alpha/\beta$ is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$$A^*x = \lambda^*B^*x$$

and if α is nonzero, $\mu = \beta/\alpha$ is an eigenvalue of the alternate form of the GNEP

$$\mu^*A^*y = B^*y.$$

Real eigenvalues (for real flavors) or the values of α and β for the i -th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

$$\alpha = S_{i, i}, \beta = P_{i, i}.$$

Input Parameters

<i>job</i>	<p>CHARACTER*1. Specifies the operations to be performed. Must be 'E' or 'S'.</p> <p>If <i>job</i> = 'E', then compute eigenvalues only;</p> <p>If <i>job</i> = 'S', then compute eigenvalues and the Schur form.</p>
<i>compq</i>	<p>CHARACTER*1. Must be 'N', 'I', or 'V'.</p> <p>If <i>compq</i> = 'N', left Schur vectors (q) are not computed;</p> <p>If <i>compq</i> = 'I', q is initialized to the unit matrix and the matrix of left Schur vectors of (H,T) is returned;</p> <p>If <i>compq</i> = 'V', q must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1^*Q is returned.</p>
<i>compz</i>	<p>CHARACTER*1. Must be 'N', 'I', or 'V'.</p> <p>If <i>compz</i> = 'N', right Schur vectors (z) are not computed;</p> <p>If <i>compz</i> = 'I', z is initialized to the unit matrix and the matrix of right Schur vectors of (H,T) is returned;</p> <p>If <i>compz</i> = 'V', z must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1^*Z is returned.</p>
<i>n</i>	<p>INTEGER. The order of the matrices H, T, Q, and Z</p> <p>($n \geq 0$).</p>
<i>ilo, ihi</i>	<p>INTEGER. <i>ilo</i> and <i>ihi</i> mark the rows and columns of H which are in Hessenberg form. It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$.</p> <p>Constraint:</p> <p>If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;</p> <p>if $n = 0$, then $ilo = 1$ and $ihi = 0$.</p>
<i>h, t, q, z, work</i>	<p>REAL for <code>shgeqz</code></p> <p>DOUBLE PRECISION for <code>dhgeqz</code></p> <p>COMPLEX for <code>chgeqz</code></p> <p>DOUBLE COMPLEX for <code>zhgeqz</code>.</p>

Arrays:

On entry, $h(ldh,*)$ contains the n -by- n upper Hessenberg matrix H .

The second dimension of h must be at least $\max(1, n)$.

On entry, $t(ldt,*)$ contains the n -by- n upper triangular matrix T .

The second dimension of t must be at least $\max(1, n)$.

$q(ldq,*)$:

On entry, if $compq = 'V'$, this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (A,B) to generalized Hessenberg form.

If $compq = 'N'$, then q is not referenced.

The second dimension of q must be at least $\max(1, n)$.

$z(ldz,*)$:

On entry, if $compz = 'V'$, this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (A,B) to generalized Hessenberg form.

If $compz = 'N'$, then z is not referenced.

The second dimension of z must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldh INTEGER. The leading dimension of h ; at least $\max(1, n)$.

ldt INTEGER. The leading dimension of t ; at least $\max(1, n)$.

ldq INTEGER. The leading dimension of q ;

If $compq = 'N'$, then $ldq \geq 1$.

If $compq = 'I'$ or $'V'$, then $ldq \geq \max(1, n)$.

ldz INTEGER. The leading dimension of z ;

If $compz = 'N'$, then $ldz \geq 1$.

If $compz = 'I'$ or $'V'$, then $ldz \geq \max(1, n)$.

$lwork$ INTEGER. The dimension of the array $work$.

$lwork \geq \max(1, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for details.

$rwork$ REAL for `chgeqz`

DOUBLE PRECISION for `zhgeqz`.

Workspace array, size at least $\max(1, n)$. Used in complex flavors only.

Output Parameters

h For real flavors:

If $job = 'S'$, then on exit h contains the upper quasi-triangular matrix S from the generalized Schur factorization.

If $job = 'E'$, then on exit the diagonal blocks of h match those of S , but the rest of h is unspecified.

For complex flavors:

If $job = 'S'$, then, on exit, h contains the upper triangular matrix S from the generalized Schur factorization.

If $job = 'E'$, then on exit the diagonal of h matches that of S , but the rest of h is unspecified.

t

If $job = 'S'$, then, on exit, t contains the upper triangular matrix P from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form, that is, if $h(j+1,j)$ is non-zero, then $t(j+1,j) = t(j,j+1) = 0$ and $t(j,j)$ and $t(j+1,j+1)$ will be positive.

If $job = 'E'$, then on exit the diagonal blocks of t match those of P , but the rest of t is unspecified.

For complex flavors:

if $job = 'E'$, then on exit the diagonal of t matches that of P , but the rest of t is unspecified.

$alphar, alphai$

REAL for shgeqz;

DOUBLE PRECISION for dhgeqz.

Arrays, size at least $\max(1, n)$. The real and imaginary parts, respectively, of each scalar $alpha$ defining an eigenvalue of GNEP.

If $alphai(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -th eigenvalues are a complex conjugate pair, with

$$alphai(j+1) = -alphai(j).$$

$alpha$

COMPLEX for chgeqz;

DOUBLE COMPLEX for zhgeqz.

Array, size at least $\max(1, n)$.

The complex scalars $alpha$ that define the eigenvalues of GNEP. $alphai(i) = S_i, i$ in the generalized Schur factorization.

$beta$

REAL for shgeqz

DOUBLE PRECISION for dhgeqz

COMPLEX for chgeqz

DOUBLE COMPLEX for zhgeqz.

Array, size at least $\max(1, n)$.

For real flavors:

The scalars $beta$ that define the eigenvalues of GNEP.

Together, the quantities $\alpha = (\alpha_r(j), \alpha_{i,j})$ and $\beta = \beta(j)$ represent the j -th eigenvalue of the matrix pair (A,B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

For complex flavors:

The real non-negative scalars β that define the eigenvalues of GNEP.

$\beta(i) = P_i$, i in the generalized Schur factorization. Together, the quantities $\alpha = \alpha(j)$ and $\beta = \beta(j)$ represent the j -th eigenvalue of the matrix pair (A,B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

q	On exit, if $compq = 'I'$, q is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (H,T) , and if $compq = 'V'$, q is overwritten by the orthogonal/unitary matrix of left Schur vectors of (A,B) .
z	On exit, if $compz = 'I'$, z is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (H,T) , and if $compz = 'V'$, z is overwritten by the orthogonal/unitary matrix of right Schur vectors of (A,B) .
$work(1)$	If $info \geq 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = 1, \dots, n$, the QZ iteration did not converge. (H,T) is not in Schur form, but $\alpha_r(i)$, $\alpha_{i,i}$ (for real flavors), $\alpha(i)$ (for complex flavors), and $\beta(i)$, $i=info+1, \dots, n$ should be correct. If $info = n+1, \dots, 2n$, the shift calculation failed. (H,T) is not in Schur form, but $\alpha_r(i)$, $\alpha_{i,i}$ (for real flavors), $\alpha(i)$ (for complex flavors), and $\beta(i)$, $i=info-n+1, \dots, n$ should be correct.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hgeqz` interface are the following:

h	Holds the matrix H of size (n,n) .
t	Holds the matrix T of size (n,n) .
α_r	Holds the vector of length n . Used in real flavors only.
$\alpha_{i,i}$	Holds the vector of length n . Used in real flavors only.
α	Holds the vector of length n . Used in complex flavors only.

<i>beta</i>	Holds the vector of length <i>n</i> .
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n,n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n,n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'E' or 'S'. The default value is 'E'.
<i>compq</i>	<p>If omitted, this argument is restored based on the presence of argument <i>q</i> as follows:</p> <p><i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted.</p> <p>If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present.</p> <p>Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.</p>
<i>compz</i>	<p>If omitted, this argument is restored based on the presence of argument <i>z</i> as follows:</p> <p><i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted.</p> <p>If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present.</p> <p>Note an error condition if <i>compz</i> is present and <i>z</i> is omitted.</p>

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

```
call stgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
```

```

call dtgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work,
info)
call ctgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work,
rwork, info)
call ztgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work,
rwork, info)
call tgevc(s, p [,howmny] [,select] [,vl] [,vr] [,m] [,info])

```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (S,P) , where S is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and P is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (A,B) :

$$A = Q^* S^* Z^H, B = Q^* P^* Z^H$$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector x and the left eigenvector y of (S,P) corresponding to an eigenvalue w are defined by:

$$S^* x = w^* P^* x, y^H S = w^* y^H P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of S and P .

This routine returns the matrices X and/or Y of right and left eigenvectors of (S,P) , or the products $Z^* X$ and/or $Q^* Y$, where Z and Q are input matrices.

If Q and Z are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (A,B) , then $Z^* X$ and $Q^* Y$ are the matrices of right and left eigenvectors of (A,B) .

Input Parameters

<i>side</i>	CHARACTER*1. Must be 'R', 'L', or 'B'. If <i>side</i> = 'R', compute right eigenvectors only. If <i>side</i> = 'L', compute left eigenvectors only. If <i>side</i> = 'B', compute both right and left eigenvectors.
<i>howmny</i>	CHARACTER*1. Must be 'A', 'B', or 'S'. If <i>howmny</i> = 'A', compute all right and/or left eigenvectors. If <i>howmny</i> = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in <i>vr</i> and/or <i>vl</i> . If <i>howmny</i> = 'S', compute selected right and/or left eigenvectors, specified by the logical array <i>select</i> .
<i>select</i>	LOGICAL. Array, size at least max (1, <i>n</i>). If <i>howmny</i> = 'S', <i>select</i> specifies the eigenvectors to be computed.

If *howmny* = 'A' or 'B', *select* is not referenced.

For real flavors:

If *w(j)* is a real eigenvalue, the corresponding real eigenvector is computed if *select(j)* is `.TRUE.`.

If *w(j)* and *omega(j + 1)* are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select(j)* or *select(j + 1)* is `.TRUE.`, and on exit *select(j)* is set to `.TRUE.` and *select(j + 1)* is set to `.FALSE.`.

For complex flavors:

The eigenvector corresponding to the *j*-th eigenvalue is computed if *select(j)* is `.TRUE.`.

n

INTEGER. The order of the matrices *S* and *P* ($n \geq 0$).

s, p, vl, vr, work

REAL for *stgevc*

DOUBLE PRECISION for *dtgevc*

COMPLEX for *ctgevc*

DOUBLE COMPLEX for *ztgevc*.

Arrays:

s(lds,)* contains the matrix *S* from a generalized Schur factorization as computed by `?hgeqz`. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

The second dimension of *s* must be at least $\max(1, n)$.

p(ldp,)* contains the upper triangular matrix *P* from a generalized Schur factorization as computed by `?hgeqz`.

For real flavors, 2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* must be in positive diagonal form.

For complex flavors, *P* must have real diagonal elements. The second dimension of *p* must be at least $\max(1, n)$.

If *side* = 'L' or 'B' and *howmny* = 'B', *vl(ldvl,*)* must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by `?hgeqz`). The second dimension of *vl* must be at least $\max(1, mm)$.

If *side* = 'R', *vl* is not referenced.

If *side* = 'R' or 'B' and *howmny* = 'B', *vr(ldvr,*)* must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by `?hgeqz`). The second dimension of *vr* must be at least $\max(1, mm)$.

If *side* = 'L', *vr* is not referenced.

work()* is a workspace array.

size at least $\max(1, 6*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

lds

INTEGER. The leading dimension of *s*; at least $\max(1, n)$.

<i>ldp</i>	INTEGER. The leading dimension of p ; at least $\max(1, n)$.
<i>ldvl</i>	INTEGER. The leading dimension of vl ; If $side = 'L'$ or $'B'$, then $ldvl \geq n$. If $side = 'R'$, then $ldvl \geq 1$.
<i>ldvr</i>	INTEGER. The leading dimension of vr ; If $side = 'R'$ or $'B'$, then $ldvr \geq n$. If $side = 'L'$, then $ldvr \geq 1$.
<i>mm</i>	INTEGER. The number of columns in the arrays vl and/or vr ($mm \geq m$).
<i>rwork</i>	REAL for <code>ctgevc</code> DOUBLE PRECISION for <code>ztgevc</code> . Workspace array, size at least $\max(1, 2*n)$. Used in complex flavors only.

Output Parameters

<i>vl</i>	<p>On exit, if $side = 'L'$ or $'B'$, vl contains:</p> <p>if $howmny = 'A'$, the matrix Y of left eigenvectors of (S,P);</p> <p>if $howmny = 'B'$, the matrix $Q*Y$;</p> <p>if $howmny = 'S'$, the left eigenvectors of (S,P) specified by <i>select</i>, stored consecutively in the columns of vl, in the same order as their eigenvalues.</p> <p><i>For real flavors:</i></p> <p>A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.</p>
<i>vr</i>	<p>On exit, if $side = 'R'$ or $'B'$, vr contains:</p> <p>if $howmny = 'A'$, the matrix X of right eigenvectors of (S,P);</p> <p>if $howmny = 'B'$, the matrix $Z*X$;</p> <p>if $howmny = 'S'$, the right eigenvectors of (S,P) specified by <i>select</i>, stored consecutively in the columns of vr, in the same order as their eigenvalues.</p> <p><i>For real flavors:</i></p> <p>A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.</p>
<i>m</i>	<p>INTEGER. The number of columns in the arrays vl and/or vr actually used to store the eigenvectors.</p> <p>If $howmny = 'A'$ or $'B'$, m is set to n.</p> <p><i>For real flavors:</i></p> <p>Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.</p> <p><i>For complex flavors:</i></p> <p>Each selected eigenvector occupies one column.</p>

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

For real flavors:

if *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgevc` interface are the following:

<i>s</i>	Holds the matrix <i>S</i> of size (<i>n</i> , <i>n</i>).
<i>p</i>	Holds the matrix <i>P</i> of size (<i>n</i> , <i>n</i>).
<i>select</i>	Holds the vector of length <i>n</i> .
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>mm</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>mm</i>).
<i>side</i>	Restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: <i>side</i> = 'B', if both <i>vl</i> and <i>vr</i> are present, <i>side</i> = 'L', if <i>vl</i> is present and <i>vr</i> omitted, <i>side</i> = 'R', if <i>vl</i> is omitted and <i>vr</i> present, Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.
<i>howmny</i>	If omitted, this argument is restored based on the presence of argument <i>select</i> as follows: <i>howmny</i> = 'S', if <i>select</i> is present, <i>howmny</i> = 'A', if <i>select</i> is omitted. If present, <i>howmny</i> must be equal to 'A' or 'B' and the argument <i>select</i> must be omitted. Note that there will be an error condition if both <i>howmny</i> and <i>select</i> are present.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

```
call stgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
```

```
call dtgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork, info)
```

```
call ctgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
```

```
call ztgeexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
call tgeexc(a, b [,ifst] [,ilst] [,z] [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A, B) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q^* (A, B) * Z^H,$$

so that the diagonal block of (A, B) with row index *ifst* is moved to row *ilst*. Matrix pair (A, B) must be in a generalized real-Schur/Schur canonical form (as returned by [gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and B is upper triangular. Optionally, the matrices Q and Z of generalized Schur vectors are updated.

$$Q_{in} * A_{in} * Z_{in}^T = Q_{out} * A_{out} * Z_{out}^T$$

$$Q_{in} * B_{in} * Z_{in}^T = Q_{out} * B_{out} * Z_{out}^T.$$

Input Parameters

wantq, wantz LOGICAL.
 If *wantq* = .TRUE., update the left transformation matrix Q ;
 If *wantq* = .FALSE., do not update Q ;
 If *wantz* = .TRUE., update the right transformation matrix Z ;
 If *wantz* = .FALSE., do not update Z .

n INTEGER. The order of the matrices A and B ($n \geq 0$).

a, b, q, z REAL for stgeexc
 DOUBLE PRECISION for dtgeexc
 COMPLEX for ctgeexc
 DOUBLE COMPLEX for ztgeexc.

Arrays:
a(lda,)* contains the matrix A .
 The second dimension of a must be at least $\max(1, n)$.
b(ldb,)* contains the matrix B . The second dimension of b must be at least $\max(1, n)$.
q(ldq,)*
 If *wantq* = .FALSE., then q is not referenced.
 If *wantq* = .TRUE., then q must contain the orthogonal/unitary matrix Q .
 The second dimension of q must be at least $\max(1, n)$.
z(ldz,)*
 If *wantz* = .FALSE., then z is not referenced.
 If *wantz* = .TRUE., then z must contain the orthogonal/unitary matrix Z .

	The second dimension of z must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
<i>ldq</i>	INTEGER. The leading dimension of q ; If <i>wantq</i> = <code>.FALSE.</code> , then $ldq \geq 1$. If <i>wantq</i> = <code>.TRUE.</code> , then $ldq \geq \max(1, n)$.
<i>ldz</i>	INTEGER. The leading dimension of z ; If <i>wantz</i> = <code>.FALSE.</code> , then $ldz \geq 1$. If <i>wantz</i> = <code>.TRUE.</code> , then $ldz \geq \max(1, n)$.
<i>ifst, ilst</i>	INTEGER. Specify the reordering of the diagonal blocks of (A, B) . The block with row index <i>ifst</i> is moved to row <i>ilst</i> , by a sequence of swapping between adjacent blocks. Constraint: $1 \leq ifst, ilst \leq n$.
<i>work</i>	REAL for <code>stgexc</code> ; DOUBLE PRECISION for <code>dtgexc</code> . Workspace array, size (<i>lwork</i>). Used in real flavors only.
<i>lwork</i>	INTEGER. The dimension of <i>work</i> ; must be at least $4n + 16$. If <i>lwork</i> = <code>-1</code> , then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for details.

Output Parameters

<i>a, b, q, z</i>	Overwritten by the updated matrices A, B, Q , and Z respectively.
<i>ifst, ilst</i>	Overwritten for real flavors only. If <i>ifst</i> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <i>ilst</i> always points to the first row of the block in its final position (which may differ from its input value by ± 1).
<i>info</i>	INTEGER. If <i>info</i> = <code>0</code> , the execution is successful. If <i>info</i> = <code>-i</code> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <code>1</code> , the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is ill-conditioned. (A, B) may have been partially reordered, and <i>ilst</i> points to the first row of the current position of the block being moved.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgexc` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>b</code>	Holds the matrix B of size (n,n) .
<code>z</code>	Holds the matrix Z of size (n,n) .
<code>q</code>	Holds the matrix Q of size (n,n) .
<code>wantq</code>	Restored based on the presence of the argument <code>q</code> as follows: <code>wantq = .TRUE</code> , if <code>q</code> is present, <code>wantq = .FALSE</code> , if <code>q</code> is omitted.
<code>wantz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>wantz = .TRUE</code> , if <code>z</code> is present, <code>wantz = .FALSE</code> , if <code>z</code> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsen

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B) .

Syntax

```
call stgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphas, alphas, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

```
call dtgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphas, alphas, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

```
call ctgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z,
ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

```
call ztgsen(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z,
ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

```
call tgsen(a, b, select [,alphas] [,alphais] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [,dif]
[,m] [,info])
```

```
call tgsen(a, b, select [,alpha] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [, dif] [,m]
[,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A, B) (in terms of an orthogonal/unitary equivalence transformation $Q^{T*} (A, B) * Z$ for real flavors or $Q^{H*} (A, B) * Z$ for complex flavors), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A, B) . The leading columns of Q and Z form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(A, B) must be in generalized real-Schur/Schur canonical form (as returned by `gges`), that is, A and B are both upper triangular.

?`tgsen` also computes the generalized eigenvalues

$$\omega_j = (\text{alphar}(j) + \text{alphai}(j)*i) / \text{beta}(j) \text{ (for real flavors)}$$

$$\omega_j = \text{alpha}(j) / \text{beta}(j) \text{ (for complex flavors)}$$

of the reordered matrix pair (A, B) .

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are `Difu[(A11, B11), (A22, B22)]` and `Difl[(A11, B11), (A22, B22)]`, that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

<code>ijob</code>	<p>INTEGER. Specifies whether condition numbers are required for the cluster of eigenvalues (pl and pr) or the deflating subspaces <code>Difu</code> and <code>Difl</code>.</p> <p>If <code>ijob = 0</code>, only reorder with respect to <code>select</code>;</p> <p>If <code>ijob = 1</code>, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (pl and pr);</p> <p>If <code>ijob = 2</code>, compute upper bounds on <code>Difu</code> and <code>Difl</code>, using F-norm-based estimate (<code>dif(1:2)</code>);</p> <p>If <code>ijob = 3</code>, compute estimate of <code>Difu</code> and <code>Difl</code>, using 1-norm-based estimate (<code>dif(1:2)</code>). This option is about 5 times as expensive as <code>ijob = 2</code>;</p> <p>If <code>ijob = 4</code>, compute pl, pr and dif (i.e., options 0, 1 and 2 above). This is an economic version to get it all;</p> <p>If <code>ijob = 5</code>, compute pl, pr and dif (i.e., options 0, 1 and 3 above).</p>
<code>wantq, wantz</code>	<p>LOGICAL.</p> <p>If <code>wantq = .TRUE.</code>, update the left transformation matrix Q;</p> <p>If <code>wantq = .FALSE.</code>, do not update Q;</p> <p>If <code>wantz = .TRUE.</code>, update the right transformation matrix Z;</p> <p>If <code>wantz = .FALSE.</code>, do not update Z.</p>
<code>select</code>	<p>LOGICAL.</p> <p>Array, size at least $\max(1, n)$. Specifies the eigenvalues in the selected cluster.</p>

To select an eigenvalue ω_j , *select(j)* must be `.TRUE.` For real flavors: to select a complex conjugate pair of eigenvalues ω_j and ω_{j+1} (corresponding 2 by 2 diagonal block), *select(j)* and/or *select(j + 1)* must be set to `.TRUE.`; the complex conjugate ω_j and ω_{j+1} must be either both included in the cluster or both excluded.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, b, q, z, work REAL for `stgsen`

DOUBLE PRECISION for `dtgsen`

COMPLEX for `ctgsen`

DOUBLE COMPLEX for `ztgsen`.

Arrays:

a(lda,)* contains the matrix *A*.

For real flavors: *A* is upper quasi-triangular, with (*A, B*) in generalized real Schur canonical form.

For complex flavors: *A* is upper triangular, in generalized Schur canonical form.

The second dimension of *a* must be at least $\max(1, n)$.

b(ldb,)* contains the matrix *B*.

For real flavors: *B* is upper triangular, with (*A, B*) in generalized real Schur canonical form.

For complex flavors: *B* is upper triangular, in generalized Schur canonical form. The second dimension of *b* must be at least $\max(1, n)$.

q(ldq,)*

If *wantq* = `.TRUE.`, then *q* is an *n*-by-*n* matrix;

If *wantq* = `.FALSE.`, then *q* is not referenced.

The second dimension of *q* must be at least $\max(1, n)$.

z(ldz,)*

If *wantz* = `.TRUE.`, then *z* is an *n*-by-*n* matrix;

If *wantz* = `.FALSE.`, then *z* is not referenced.

The second dimension of *z* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

ldq INTEGER. The leading dimension of *q*; $ldq \geq 1$.

If *wantq* = `.TRUE.`, then $ldq \geq \max(1, n)$.

ldz INTEGER. The leading dimension of *z*; $ldz \geq 1$.

If *wantz* = `.TRUE.`, then $ldz \geq \max(1, n)$.

lwork INTEGER. The dimension of the array *work*.

For real flavors:

If $ijob = 1, 2, \text{ or } 4$, $lwork \geq \max(4n+16, 2m(n-m))$.

If $ijob = 3 \text{ or } 5$, $lwork \geq \max(4n+16, 4m(n-m))$.

For complex flavors:

If $ijob = 1, 2, \text{ or } 4$, $lwork \geq \max(1, 2m(n-m))$.

If $ijob = 3 \text{ or } 5$, $lwork \geq \max(1, 4m(n-m))$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER. The dimension of the array *iwork*.

For real flavors:

If $ijob = 1, 2, \text{ or } 4$, $liwork \geq n+6$.

If $ijob = 3 \text{ or } 5$, $liwork \geq \max(n+6, 2m(n-m))$.

For complex flavors:

If $ijob = 1, 2, \text{ or } 4$, $liwork \geq n+2$.

If $ijob = 3 \text{ or } 5$, $liwork \geq \max(n+2, 2m(n-m))$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a, b Overwritten by the reordered matrices *A* and *B*, respectively.

alphar, alphai REAL for stgsen;
DOUBLE PRECISION for dtgsen.
Arrays, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in real flavors.
See *beta*.

alpha COMPLEX for ctgsen;
DOUBLE COMPLEX for ztgsen.
Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors.
See *beta*.

beta REAL for stgsen
DOUBLE PRECISION for dtgsen
COMPLEX for ctgsen

DOUBLE COMPLEX for ztgsen.

Array, size at least $\max(1, n)$.

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

$\text{alphar}(j) + \text{alphai}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations.

If $\text{alphai}(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j + 1)$ -st eigenvalues are a complex conjugate pair, with $\text{alphai}(j + 1)$ negative.

For complex flavors:

The diagonal elements of A and B , respectively, when the pair (A, B) has been reduced to generalized Schur form. $\text{alpha}(i)/\text{beta}(i)$, $i=1, \dots, n$ are the generalized eigenvalues.

q If $\text{wantq} = \text{.TRUE.}$, then, on exit, Q has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B) . The leading m columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

z If $\text{wantz} = \text{.TRUE.}$, then, on exit, Z has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B) . The leading m columns of Z form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

m INTEGER.

The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \leq m \leq n$.

pl, pr REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If $i\text{job} = 1, 4$, or 5 , pl and pr are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster.

$0 < pl, pr \leq 1$. If $m = 0$ or $m = n$, $pl = pr = 1$.

If $i\text{job} = 0, 2$ or 3 , pl and pr are not referenced

dif REAL for single precision flavors; DOUBLE PRECISION for double precision flavors.

Array, size (2).

If $i\text{job} \geq 2$, $dif(1:2)$ store the estimates of $Difu$ and $Difl$.

If $i\text{job} = 2$ or 4 , $dif(1:2)$ are F-norm-based upper bounds on $Difu$ and $Difl$.

If $i\text{job} = 3$ or 5 , $dif(1:2)$ are 1-norm-based estimates of $Difu$ and $Difl$.

If $m = 0$ or $m = n$, $dif(1:2) = \text{F-norm}([A, B])$.

If $ijob = 0$ or 1 , dif is not referenced.

$work(1)$ If $ijob$ is not 0 and $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$iwork(1)$ If $ijob$ is not 0 and $info = 0$, on exit, $iwork(1)$ contains the minimum value of $liwork$ required for optimum performance. Use this $liwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, Reordering of (A, B) failed because the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is very ill-conditioned. (A, B) may have been partially reordered.

If $ijob > 0$, 0 is returned in dif , pl and pr .

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgseu` interface are the following:

a	Holds the matrix A of size (n,n) .
b	Holds the matrix B of size (n,n) .
$select$	Holds the vector of length n .
$alphan$	Holds the vector of length n . Used in real flavors only.
$alphai$	Holds the vector of length n . Used in real flavors only.
$alpha$	Holds the vector of length n . Used in complex flavors only.
$beta$	Holds the vector of length n .
q	Holds the matrix Q of size (n,n) .
z	Holds the matrix Z of size (n,n) .
dif	Holds the vector of length (2) .
$ijob$	Must be $0, 1, 2, 3, 4$, or 5 . The default value is 0 .
$wantq$	Restored based on the presence of the argument q as follows: $wantq = .TRUE$, if q is present, $wantq = .FALSE$, if q is omitted.
$wantz$	Restored based on the presence of the argument z as follows: $wantz = .TRUE$, if z is present,

`wantz = .FALSE`, if `z` is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsyl

Solves the generalized Sylvester equation.

Syntax

```
call stgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale, dif, work, lwork, iwork, info)
```

```
call dtgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale, dif, work, lwork, iwork, info)
```

```
call ctgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale, dif, work, lwork, iwork, info)
```

```
call ztgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale, dif, work, lwork, iwork, info)
```

```
call tgsyl(a, b, c, d, e, f [,ijob] [,trans] [,scale] [,dif] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves the generalized Sylvester equation:

$$A^*R - L^*B = scale^*C$$

$$D^*R - L^*E = scale^*F$$

where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively, with real/complex entries. (A, D) and (B, E) must be in generalized real-Schur/Schur canonical form, that is, A, B are upper quasi-triangular/triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F) . The factor `scale`, $0 \leq scale \leq 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve $Z^*x = scale^*b$, where Z is defined as

$$Z = \begin{pmatrix} \text{kron}(I_n, A) & -\text{kron}(B^T, I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E^T, I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size k and X^T is the transpose/conjugate-transpose of X . $kron(X, Y)$ is the Kronecker product between the matrices X and Y .

If $trans = 'T'$ (for real flavors), or $trans = 'C'$ (for complex flavors), the routine `?tgstyl` solves the transposed/conjugate-transposed system $Z^T * y = scale * b$, which is equivalent to solve for R and L in

$$A^T * R + D^T * L = scale * C$$

$$R * B^T + L * E^T = scale * (-F)$$

This case ($trans = 'T'$ for `stgsyl/dtgsyl` or $trans = 'C'$ for `ctgsyl/ztgsyl`) is used to compute an one-norm-based estimate of `Dif[(A, D), (B, E)]`, the separation between the matrix pairs (A, D) and (B, E) , using `lacon/lacon`.

If $ijob \geq 1$, `?tgstyl` computes a Frobenius norm-based estimate of `Dif[(A, D), (B, E)]`. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z . This is a level 3 BLAS algorithm.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N', 'T', or 'C'. If $trans = 'N'$, solve the generalized Sylvester equation. If $trans = 'T'$, solve the 'transposed' system (for real flavors only). If $trans = 'C'$, solve the 'conjugate transposed' system (for complex flavors only).
<i>ijob</i>	INTEGER. Specifies what kind of functionality to be performed: If $ijob = 0$, solve the generalized Sylvester equation only; If $ijob = 1$, perform the functionality of $ijob = 0$ and $ijob = 3$; If $ijob = 2$, perform the functionality of $ijob = 0$ and $ijob = 4$; If $ijob = 3$, only an estimate of <code>Dif[(A, D), (B, E)]</code> is computed (look ahead strategy is used); If $ijob = 4$, only an estimate of <code>Dif[(A, D), (B, E)]</code> is computed (<code>?gecon</code> on sub-systems is used). If $trans = 'T'$ or 'C', $ijob$ is not referenced.
<i>m</i>	INTEGER. The order of the matrices A and D , and the row dimension of the matrices C, F, R and L .
<i>n</i>	INTEGER. The order of the matrices B and E , and the column dimension of the matrices C, F, R and L .
<i>a, b, c, d, e, f, work</i>	REAL for <code>stgsyl</code> DOUBLE PRECISION for <code>dtgsyl</code> COMPLEX for <code>ctgsyl</code> DOUBLE COMPLEX for <code>ztgsyl</code> . Arrays: <i>a(lda,*)</i> contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix A . The second dimension of a must be at least <code>max(1, m)</code> .

$b(ldb,*)$ contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix B . The second dimension of b must be at least $\max(1, n)$.

$c ldc,*)$ contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by *trans*)

The second dimension of c must be at least $\max(1, n)$.

$d(ldd,*)$ contains the upper triangular matrix D .

The second dimension of d must be at least $\max(1, m)$.

$e(lde,*)$ contains the upper triangular matrix E .

The second dimension of e must be at least $\max(1, n)$.

$f(ldf,*)$ contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by *trans*)

The second dimension of f must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, n)$.

ldc INTEGER. The leading dimension of c ; at least $\max(1, m)$.

ldd INTEGER. The leading dimension of d ; at least $\max(1, m)$.

lde INTEGER. The leading dimension of e ; at least $\max(1, n)$.

ldf INTEGER. The leading dimension of f ; at least $\max(1, m)$.

lwork INTEGER.

The dimension of the array *work*. $lwork \geq 1$.

If $ijob = 1$ or 2 and $trans = 'N'$, $lwork \geq \max(1, 2*m*n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, size at least $(m+n+6)$ for real flavors, and at least $(m+n+2)$ for complex flavors.

Output Parameters

c If $ijob=0, 1, \text{ or } 2$, overwritten by the solution R .
If $ijob=3$ or 4 and $trans = 'N'$, c holds R , the solution achieved during the computation of the Dif-estimate.

f If $ijob=0, 1, \text{ or } 2$, overwritten by the solution L .
If $ijob=3$ or 4 and $trans = 'N'$, f holds L , the solution achieved during the computation of the Dif-estimate.

dif REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, *dif* is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, *dif* is an upper bound of $\text{Dif}[(A, D), (B, E)] = \sigma_{\min}(Z)$, where *Z* as defined in the description.

If *ijob* = 0, or *trans* = 'T' (for real flavors), or *trans* = 'C' (for complex flavors), *dif* is not touched.

scale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, *scale* is the scaling factor in the generalized Sylvester equation.

If $0 < \text{scale} < 1$, *c* and *f* hold the solutions *R* and *L*, respectively, to a slightly perturbed system but the input matrices *A*, *B*, *D* and *E* have not been changed.

If *scale* = 0, *c* and *f* hold the solutions *R* and *L*, respectively, to the homogeneous system with $C = F = 0$. Normally, *scale* = 1.

work(1)

If *info* = 0, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, (*A*, *D*) and (*B*, *E*) have common or close eigenvalues.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgsyl` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>m</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m</i> , <i>n</i>).
<i>d</i>	Holds the matrix <i>D</i> of size (<i>m</i> , <i>m</i>).
<i>e</i>	Holds the matrix <i>E</i> of size (<i>n</i> , <i>n</i>).
<i>f</i>	Holds the matrix <i>F</i> of size (<i>m</i> , <i>n</i>).
<i>ijob</i>	Must be 0, 1, 2, 3, or 4. The default value is 0.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

```
call stgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm,
m, work, lwork, iwork, info)
```

```
call dtgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm,
m, work, lwork, iwork, info)
```

```
call ctgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm,
m, work, lwork, iwork, info)
```

```
call ztgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm,
m, work, lwork, iwork, info)
```

```
call tgsna(a, b [,s] [,dif] [,vl] [,vr] [,select] [,m] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The real flavors *stgsna/dtgsna* of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair $(Q^*A^*Z^T, Q^*B^*Z^T)$ with orthogonal matrices Q and Z).

(A, B) must be in generalized real Schur form (as returned by [gges/gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors *ctgsna/ztgsna* estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) . (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Input Parameters

<i>job</i>	CHARACTER*1. Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'. If <i>job</i> = 'E', for eigenvalues only (compute <i>s</i>). If <i>job</i> = 'V', for eigenvectors only (compute <i>dif</i>). If <i>job</i> = 'B', for both eigenvalues and eigenvectors (compute both <i>s</i> and <i>dif</i>).
<i>howmny</i>	CHARACTER*1. Must be 'A' or 'S'.

If *howmny* = 'A', compute condition numbers for all eigenpairs.

If *howmny* = 'S', compute condition numbers for selected eigenpairs specified by the logical array *select*.

select

LOGICAL.

Array, size at least $\max(1, n)$.

If *howmny* = 'S', *select* specifies the eigenpairs for which condition numbers are required.

If *howmny* = 'A', *select* is not referenced.

For real flavors:

To select condition numbers for the eigenpair corresponding to a real eigenvalue ω_j , *select(j)* must be set to `.TRUE.`; to select condition numbers corresponding to a complex conjugate pair of eigenvalues ω_j and ω_{j+1} , either *select(j)* or *select(j + 1)* must be set to `.TRUE.`

For complex flavors:

To select condition numbers for the corresponding *j*-th eigenvalue and/or eigenvector, *select(j)* must be set to `.TRUE.`

n

INTEGER. The order of the square matrix pair (*A*, *B*)

($n \geq 0$).

a, *b*, *vl*, *vr*, *work*

REAL for *stgsna*

DOUBLE PRECISION for *dtgsna*

COMPLEX for *ctgsna*

DOUBLE COMPLEX for *ztgsna*.

Arrays:

a(lda,)* contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A* in the pair (*A*, *B*).

The second dimension of *a* must be at least $\max(1, n)$.

b(ldb,)* contains the upper triangular matrix *B* in the pair (*A*, *B*). The second dimension of *b* must be at least $\max(1, n)$.

If *job* = 'E' or 'B', *vl(ldvl,*)* must contain left eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by `tgevc`.

If *job* = 'V', *vl* is not referenced. The second dimension of *vl* must be at least $\max(1, m)$.

If *job* = 'E' or 'B', *vr(ldvr,*)* must contain right eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by `tgevc`.

If *job* = 'V', *vr* is not referenced. The second dimension of *vr* must be at least $\max(1, m)$.

work is a workspace array, its dimension $\max(1, lwork)$.

	If $job = 'E'$, $work$ is not referenced.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
$ldvl$	INTEGER. The leading dimension of vl ; $ldvl \geq 1$. If $job = 'E'$ or $'B'$, then $ldvl \geq \max(1, n)$.
$ldvr$	INTEGER. The leading dimension of vr ; $ldvr \geq 1$. If $job = 'E'$ or $'B'$, then $ldvr \geq \max(1, n)$.
mm	INTEGER. The number of elements in the arrays s and dif ($mm \geq m$).
$lwork$	INTEGER. The dimension of the array $work$. $lwork \geq \max(1, n)$. If $job = 'V'$ or $'B'$, $lwork \geq 2*n*(n+2)+16$ for real flavors, and $lwork \geq \max(1, 2*n*n)$ for complex flavors. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for details.
$iwork$	INTEGER. Workspace array, size at least $(n+6)$ for real flavors, and at least $(n+2)$ for complex flavors. If $job = 'E'$, $iwork$ is not referenced.

Output Parameters

s	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, size (mm). If $job = 'E'$ or $'B'$, contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array. If $job = 'V'$, s is not referenced. <i>For real flavors:</i> For a complex conjugate pair of eigenvalues two consecutive elements of s are set to the same value. Thus, $s(j)$, $dif(j)$, and the j -th columns of vl and vr all correspond to the same eigenpair (but not in general the j -th eigenpair, unless all eigenpairs are selected).
dif	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Array, size (mm). If $job = 'V'$ or $'B'$, contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array.

If the eigenvalues cannot be reordered to compute $dif(j)$, $dif(j)$ is set to 0; this can only occur when the true value would be very small anyway.

If $job = 'E'$, dif is not referenced.

For real flavors:

For a complex eigenvector, two consecutive elements of dif are set to the same value.

For complex flavors:

For each eigenvalue/vector specified by $select$, dif stores a Frobenius norm-based estimate of $Difl$.

m INTEGER. The number of elements in the arrays s and dif used to store the specified condition numbers; for each selected eigenvalue one element is used.

If $howmny = 'A'$, m is set to n .

work(1) *work(1)*

If job is not 'E' and $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgssna` interface are the following:

<i>a</i>	Holds the matrix A of size (n,n) .
<i>b</i>	Holds the matrix B of size (n,n) .
<i>s</i>	Holds the vector of length (mm) .
<i>dif</i>	Holds the vector of length (mm) .
<i>vl</i>	Holds the matrix VL of size (n,mm) .
<i>vr</i>	Holds the matrix VR of size (n,mm) .
<i>select</i>	Holds the vector of length n .
<i>howmny</i>	Restored based on the presence of the argument <i>select</i> as follows: $howmny = 'S'$, if <i>select</i> is present, $howmny = 'A'$, if <i>select</i> is omitted.

job Restored based on the presence of arguments *s* and *dif* as follows: *job* = 'B', if both *s* and *dif* are present, *job* = 'E', if *s* is present and *dif* omitted, *job* = 'V', if *s* is omitted and *dif* present, Note that there will be an error condition if both *s* and *dif* are omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices *A* and *B* as

$$U^H A Q = D_1 * (0 \ R),$$

$$V^H B Q = D_2 * (0 \ R),$$

where *U*, *V*, and *Q* are orthogonal/unitary matrices, *R* is a nonsingular upper triangular matrix, and *D*₁, *D*₂ are "diagonal" matrices of the structure detailed in the routines description section.

Table "Computational Routines for Generalized Singular Value Decomposition" lists LAPACK routines (FORTRAN 77 interface) that perform generalized singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
ggsvp	Computes the preprocessing decomposition for the generalized SVD
tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine [ggsvd](#) to find the GSVD of a pair of general rectangular matrices.

?ggsvp

Computes the preprocessing decomposition for the generalized SVD.

Syntax

```
call sggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, info)
```

```
call dggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, info)
```

```
call cggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, info)
```

```
call zggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, info)
```

```
call ggsvp(a, b, tola, tolb [, k] [,l] [,u] [,v] [,q] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes orthogonal matrices U , V and Q such that

$$A = U \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} V^H$$

$$= \begin{matrix} & n-k-l & k & l \\ & k & & \\ m-k & \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} & & \end{matrix}, \text{ if } m-k-l < 0$$

$$A = U \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} V^H$$

where the k -by- k matrix A_{12} and l -by- l matrix A_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$ -by- n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine [?tgsja](#).

Input Parameters

- jobu* CHARACTER*1. Must be 'U' or 'N'.
 If *jobu* = 'U', orthogonal/unitary matrix U is computed.
 If *jobu* = 'N', U is not computed.
- jobv* CHARACTER*1. Must be 'V' or 'N'.
 If *jobv* = 'V', orthogonal/unitary matrix V is computed.

If $jobv = 'N'$, V is not computed.

jobq CHARACTER*1. Must be 'Q' or 'N'.
 If $jobq = 'Q'$, orthogonal/unitary matrix Q is computed.
 If $jobq = 'N'$, Q is not computed.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

p INTEGER. The number of rows of the matrix B ($p \geq 0$).

n INTEGER. The number of columns of the matrices A and B ($n \geq 0$).

a, b, tau, work REAL for *sggsvp*
 DOUBLE PRECISION for *dggsvp*
 COMPLEX for *cggsvp*
 DOUBLE COMPLEX for *zggsvp*.

Arrays:
a(lda,)* contains the m -by- n matrix A .
 The second dimension of a must be at least $\max(1, n)$.
b(ldb,)* contains the p -by- n matrix B .
 The second dimension of b must be at least $\max(1, n)$.
tau()* is a workspace array.
 The dimension of tau must be at least $\max(1, n)$.
work()* is a workspace array.
 The dimension of $work$ must be at least $\max(1, 3n, m, p)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, p)$.

tola, tolb REAL for single-precision flavors
 DOUBLE PRECISION for double-precision flavors.
tola and *tolb* are the thresholds to determine the effective numerical rank of matrix B and a subblock of A . Generally, they are set to
 $tola = \max(m, n) * ||A|| * \text{MACHEPS}$,
 $tolb = \max(p, n) * ||B|| * \text{MACHEPS}$.
 The size of *tola* and *tolb* may affect the size of backward errors of the decomposition.

ldu INTEGER. The leading dimension of the output array u . $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.

ldv INTEGER. The leading dimension of the output array v . $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.

<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> . $ldq \geq \max(1, n)$ if <i>jobq</i> = 'Q'; $ldq \geq 1$ otherwise.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cggsvp</i> DOUBLE PRECISION for <i>zggsvp</i> . Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>a</i>	Overwritten by the triangular (or trapezoidal) matrix described in the <i>Description</i> section.
<i>b</i>	Overwritten by the triangular matrix described in the <i>Description</i> section.
<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of subblocks. The sum $k + l$ is equal to effective numerical rank of $(A^H, B^H)^H$.
<i>u, v, q</i>	REAL for <i>sggsvp</i> DOUBLE PRECISION for <i>dggsvp</i> COMPLEX for <i>cggsvp</i> DOUBLE COMPLEX for <i>zggsvp</i> . Arrays: If <i>jobu</i> = 'U', <i>u(ldu,*)</i> contains the orthogonal/unitary matrix <i>U</i> . The second dimension of <i>u</i> must be at least $\max(1, m)$. If <i>jobu</i> = 'N', <i>u</i> is not referenced. If <i>jobv</i> = 'V', <i>v(ldv,*)</i> contains the orthogonal/unitary matrix <i>V</i> . The second dimension of <i>v</i> must be at least $\max(1, m)$. If <i>jobv</i> = 'N', <i>v</i> is not referenced. If <i>jobq</i> = 'Q', <i>q(ldq,*)</i> contains the orthogonal/unitary matrix <i>Q</i> . The second dimension of <i>q</i> must be at least $\max(1, n)$. If <i>jobq</i> = 'N', <i>q</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *ggsvp* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
----------	---

<i>b</i>	Holds the matrix <i>B</i> of size (p,n) .
<i>u</i>	Holds the matrix <i>U</i> of size (m,m) .
<i>v</i>	Holds the matrix <i>V</i> of size (p,m) .
<i>q</i>	Holds the matrix <i>Q</i> of size (n,n) .
<i>jobu</i>	Restored based on the presence of the argument <i>u</i> as follows: <i>jobu</i> = 'U', if <i>u</i> is present, <i>jobu</i> = 'N', if <i>u</i> is omitted.
<i>jobv</i>	Restored based on the presence of the argument <i>v</i> as follows: <i>jobz</i> = 'V', if <i>v</i> is present, <i>jobz</i> = 'N', if <i>v</i> is omitted.
<i>jobq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>jobz</i> = 'Q', if <i>q</i> is present, <i>jobz</i> = 'N', if <i>q</i> is omitted.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

```
call stgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

```
call dtgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

```
call ctgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

```
call ztgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

```
call tgsja(a, b, tola, tolb, k, l [,u] [,v] [,q] [,jobu] [,jobv] [,jobq] [,alpha]
[,beta] [,ncycle] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices *A* and *B*. On entry, it is assumed that matrices *A* and *B* have the following forms, which may be obtained by the preprocessing subroutine [ggsvp](#) from a general *m*-by-*n* matrix *A* and *p*-by-*n* matrix *B*:

$$A = \begin{matrix} & n-k-l & k & l \\ & k & \begin{pmatrix} 0 & A_{12} & A_{13} \\ l & 0 & A_{23} \\ m-k-l & 0 & 0 \end{pmatrix} & \end{matrix}, \text{ if } m-k-l \geq 0$$

.....

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal.

On exit,

$$U^H * A * Q = D_1 * (0 \ R), \quad V^H * B * Q = D_2 * (0 \ R),$$

where U , V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{matrix} & k & l \\ k & \begin{pmatrix} I & 0 \\ l & C \\ m-k-l & 0 & 0 \end{pmatrix} & \end{matrix}$$

$$D_2 = \begin{matrix} & k & l \\ & l & \begin{pmatrix} 0 & S \\ p-l & 0 \end{pmatrix} & \end{matrix}$$

The computation of the orthogonal/unitary transformation matrices U , V or Q is optional. These matrices may either be formed explicitly, or they *may* be postmultiplied into input matrices U_1 , V_1 , or Q_1 .

Input Parameters

<i>jobu</i>	<p>CHARACTER*1. Must be 'U', 'I', or 'N'.</p> <p>If <i>jobu</i> = 'U', <i>u</i> must contain an orthogonal/unitary matrix U_1 on entry.</p> <p>If <i>jobu</i> = 'I', <i>u</i> is initialized to the unit matrix.</p> <p>If <i>jobu</i> = 'N', <i>u</i> is not computed.</p>
<i>jobv</i>	<p>CHARACTER*1. Must be 'V', 'I', or 'N'.</p> <p>If <i>jobv</i> = 'V', <i>v</i> must contain an orthogonal/unitary matrix V_1 on entry.</p> <p>If <i>jobv</i> = 'I', <i>v</i> is initialized to the unit matrix.</p> <p>If <i>jobv</i> = 'N', <i>v</i> is not computed.</p>
<i>jobq</i>	<p>CHARACTER*1. Must be 'Q', 'I', or 'N'.</p> <p>If <i>jobq</i> = 'Q', <i>q</i> must contain an orthogonal/unitary matrix Q_1 on entry.</p> <p>If <i>jobq</i> = 'I', <i>q</i> is initialized to the unit matrix.</p> <p>If <i>jobq</i> = 'N', <i>q</i> is not computed.</p>
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>p</i>	INTEGER. The number of rows of the matrix B ($p \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
<i>k, l</i>	INTEGER. Specify the subblocks in the input matrices A and B , whose GSVD is computed.
<i>a, b, u, v, q, work</i>	<p>REAL for stgsja</p> <p>DOUBLE PRECISION for dtgsja</p> <p>COMPLEX for ctgsja</p> <p>DOUBLE COMPLEX for ztgsja.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> contains the m-by-n matrix A.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>b(ldb,*)</i> contains the p-by-n matrix B.</p> <p>The second dimension of <i>b</i> must be at least $\max(1, n)$.</p> <p>If <i>jobu</i> = 'U', <i>u(ldu,*)</i> must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).</p> <p>The second dimension of <i>u</i> must be at least $\max(1, m)$.</p> <p>If <i>jobv</i> = 'V', <i>v(ldv,*)</i> must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).</p> <p>The second dimension of <i>v</i> must be at least $\max(1, p)$.</p>

If $jobq = 'Q'$, $q(ldq,*)$ must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by `?ggsvp`).

The second dimension of q must be at least $\max(1, n)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, 2n)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, p)$.

ldu INTEGER. The leading dimension of the array u .
 $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.

ldv INTEGER. The leading dimension of the array v .
 $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.

ldq INTEGER. The leading dimension of the array q .
 $ldq \geq \max(1, n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.

$tola, tolb$ REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

$tola$ and $tolb$ are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in `?ggsvp`:

$tola = \max(m, n) * |A| * \text{MACHEPS}$,

$tolb = \max(p, n) * |B| * \text{MACHEPS}$.

Output Parameters

a On exit, $a(n-k+1:n, 1:\min(k+l, m))$ contains the triangular matrix R or part of R .

b On exit, if necessary, $b(m-k+1:l, n+m-k-l+1:n)$ contains a part of R .

$alpha, beta$ REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, size at least $\max(1, n)$. Contain the generalized singular value pairs of A and B :

$alpha(1:k) = 1$,

$beta(1:k) = 0$,

and if $m-k-l \geq 0$,

$alpha(k+1:k+l) = \text{diag}(C)$,

$beta(k+1:k+l) = \text{diag}(S)$,

or if $m-k-l < 0$,

$alpha(k+1:m) = \text{diag}(C)$, $alpha(m+1:k+l) = 0$

$beta(k+1:m) = \text{diag}(S)$,

$beta(m+1:k+1) = 1.$

Furthermore, if $k+1 < n,$

$alpha(k+1+1:n) = 0$ and

$beta(k+1+1:n) = 0.$

u If *jobu* = 'I', *u* contains the orthogonal/unitary matrix *U*.
 If *jobu* = 'U', *u* contains the product U_1*U .
 If *jobu* = 'N', *u* is not referenced.

v If *jobv* = 'I', *v* contains the orthogonal/unitary matrix *U*.
 If *jobv* = 'V', *v* contains the product V_1*V .
 If *jobv* = 'N', *v* is not referenced.

q If *jobq* = 'I', *q* contains the orthogonal/unitary matrix *U*.
 If *jobq* = 'Q', *q* contains the product Q_1*Q .
 If *jobq* = 'N', *q* is not referenced.

ncycle INTEGER. The number of cycles required for convergence.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = 1, the procedure does not converge after MAXIT cycles.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `tgsja` interface are the following:

a Holds the matrix *A* of size (*m*,*n*).

b Holds the matrix *B* of size (*p*,*n*).

u Holds the matrix *U* of size (*m*,*m*).

v Holds the matrix *V* of size (*p*,*p*).

q Holds the matrix *Q* of size (*n*,*n*).

alpha Holds the vector of length *n*.

beta Holds the vector of length *n*.

jobu If omitted, this argument is restored based on the presence of argument *u* as follows:
jobu = 'U', if *u* is present,
jobu = 'N', if *u* is omitted.

If present, *jobu* must be equal to 'I' or 'U' and the argument *u* must also be present.

Note that there will be an error condition if *jobu* is present and *u* omitted.

jobv

If omitted, this argument is restored based on the presence of argument *v* as follows:

jobv = 'V', if *v* is present,

jobv = 'N', if *v* is omitted.

If present, *jobv* must be equal to 'I' or 'V' and the argument *v* must also be present.

Note that there will be an error condition if *jobv* is present and *v* omitted.

jobq

If omitted, this argument is restored based on the presence of argument *q* as follows:

jobq = 'Q', if *q* is present,

jobq = 'N', if *q* is omitted.

If present, *jobq* must be equal to 'I' or 'Q' and the argument *q* must also be present.

Note that there will be an error condition if *jobq* is present and *q* omitted.

Cosine-Sine Decomposition

This section describes LAPACK computational routines for computing the *cosine-sine decomposition* (CS decomposition) of a partitioned unitary/orthogonal matrix. The algorithm computes a complete 2-by-2 CS decomposition, which requires simultaneous diagonalization of all the four blocks of a unitary/orthogonal matrix partitioned into a 2-by-2 block structure.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Computational Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form	bbcsd/bbcsd	bbcsd/bbcsd
Simultaneously bidiagonalize the blocks of a partitioned orthogonal matrix	orbdb unbdb	
Simultaneously bidiagonalize the blocks of a partitioned unitary matrix		orbdb unbdb

See Also

[Cosine-Sine Decomposition](#)

?bbcscd

Computes the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form.

Syntax

```
call sbbcscd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, work,
lwork, info )
```

```
call dbbcscd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, work,
lwork, info )
```

```
call cbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )
```

```
call zbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )
```

```
call bbcscd( theta, phi, u1, u2, v1t, v2t[, b11d][, b11e][, b12d][, b12e][, b21d][, b21e][, b22d]
[, b22e][, jobu1][, jobu2][, jobv1t][, jobv2t][, trans][, info] )
```

Include Files

- mkl.fi, lapack.f90

Description

mkl_lapack.fi The routine ?bbcscd computes the CS decomposition of an orthogonal or unitary matrix in bidiagonal-block form:

$$X = \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} = \begin{pmatrix} u_1 & | & \\ & & u_2 \end{pmatrix} \begin{pmatrix} C & | & -S & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ S & | & C & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} v_1 & | & \\ & & v_2 \end{pmatrix}^T$$

or

$$X = \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} = \begin{pmatrix} u_1 & | & \\ & & u_2 \end{pmatrix} \begin{pmatrix} C & | & -S & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ S & | & C & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} v_1 & | & \\ & & v_2 \end{pmatrix}^H$$

respectively.

x is m -by- m with the top-left block p -by- q . Note that q must not be larger than p , $m-p$, or $m-q$. If q is not the smallest index, x must be transposed and/or permuted in constant time using the *trans* option. See ?orcsd/?uncsd for details.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are represented implicitly by angles $theta(1:q)$ and $phi(1:q-1)$.

The orthogonal/unitary matrices u_1 , u_2 , v_1^t , and v_2^t are input/output. The input matrices are pre- or post-multiplied by the appropriate singular vector matrices.

Input Parameters

<i>jobu1</i>	CHARACTER. If equals <i>Y</i> , then u_1 is updated. Otherwise, u_1 is not updated.
<i>jobu2</i>	CHARACTER. If equals <i>Y</i> , then u_2 is updated. Otherwise, u_2 is not updated.
<i>jobv1t</i>	CHARACTER. If equals <i>Y</i> , then v_1^t is updated. Otherwise, v_1^t is not updated.
<i>jobv2t</i>	CHARACTER. If equals <i>Y</i> , then v_2^t is updated. Otherwise, v_2^t is not updated.
<i>trans</i>	CHARACTER = 'T': x , u_1 , u_2 , v_1^t , v_2^t are stored in row-major order. otherwise x , u_1 , u_2 , v_1^t , v_2^t are stored in column-major order.
<i>m</i>	INTEGER. The number of rows and columns of the orthogonal/unitary matrix X in bidiagonal-block form.
<i>p</i>	INTEGER. The number of rows in the top-left block of x . $0 \leq p \leq m$. \leq
<i>q</i>	INTEGER. The number of columns in the top-left block of x . $0 \leq q \leq \min(p, m-p, m-q)$.
<i>theta</i>	REAL for sbbcsd DOUBLE PRECISION for dbbcsd COMPLEX for cbbcsd DOUBLE COMPLEX for zbbcsd Array, size (<i>q</i>). On entry, the angles $\theta(1), \dots, \theta(q)$ that, along with $\phi(1), \dots, \phi(q-1)$, define the matrix in bidiagonal-block form as returned by orbdb/unbdb .
<i>phi</i>	REAL for sbbcsd DOUBLE PRECISION for dbbcsd COMPLEX for cbbcsd DOUBLE COMPLEX for zbbcsd Array, size (<i>q-1</i>). The angles $\phi(1), \dots, \phi(q-1)$ that, along with $\theta(1), \dots, \theta(q)$, define the matrix in bidiagonal-block form as returned by orbdb/unbdb .
<i>u1</i>	REAL for sbbcsd DOUBLE PRECISION for dbbcsd COMPLEX for cbbcsd

DOUBLE COMPLEX for `zbbcsd`
Array, size $(ldu1, p)$.
On entry, a p -by- p matrix.

`ldu1` INTEGER. The leading dimension of the array u_1 , $ldu1 \leq \max(1, p)$.

`u2` REAL for `sbbcsd`
DOUBLE PRECISION for `dbbcsd`
COMPLEX for `cbbcsd`
DOUBLE COMPLEX for `zbbcsd`
Array, size $(ldu2, m-p)$.
On entry, an $(m-p)$ -by- $(m-p)$ matrix.

`ldu2` INTEGER. The leading dimension of the array u_2 , $ldu2 \leq \max(1, m-p)$.

`v1t` REAL for `sbbcsd`
DOUBLE PRECISION for `dbbcsd`
COMPLEX for `cbbcsd`
DOUBLE COMPLEX for `zbbcsd`
Array, size $(ldv1t, q)$.
On entry, a q -by- q matrix.

`ldv1t` INTEGER. The leading dimension of the array $v1t$, $ldv1t \leq \max(1, q)$.

`v2t` REAL for `sbbcsd`
DOUBLE PRECISION for `dbbcsd`
COMPLEX for `cbbcsd`
DOUBLE COMPLEX for `zbbcsd`
Array, size $(ldv2t, m-q)$.
On entry, an $(m-q)$ -by- $(m-q)$ matrix.

`ldv2t` INTEGER. The leading dimension of the array $v2t$, $ldv2t \leq \max(1, m-q)$.

`work` REAL for `sbbcsd`
DOUBLE PRECISION for `dbbcsd`
COMPLEX for `cbbcsd`
DOUBLE COMPLEX for `zbbcsd`
Workspace array, size $(\max(1, lwork))$.

`lwork` INTEGER. The size of the `work` array. $lwork \geq \max(1, 8 * q)$
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

Output Parameters

<i>theta</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>On exit, the angles whose cosines and sines define the diagonal blocks in the CS decomposition.</p>
<i>u1</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>On exit, <i>u1</i> is postmultiplied by the left singular vector matrix common to $[b_{11} ; 0]$ and $[b_{12} \ 0 \ 0 ; 0 \ -I \ 0]$.</p>
<i>u2</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>On exit, <i>u2</i> is postmultiplied by the left singular vector matrix common to $[b_{21} ; 0]$ and $[b_{22} \ 0 \ 0 ; 0 \ 0 \ I]$.</p>
<i>v1t</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>Array, size (<i>q</i>).</p> <p>On exit, <i>v1t</i> is premultiplied by the transpose of the right singular vector matrix common to $[b_{11} ; 0]$ and $[b_{21} ; 0]$.</p>
<i>v2t</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>On exit, <i>v2t</i> is premultiplied by the transpose of the right singular vector matrix common to $[b_{12} \ 0 \ 0 ; 0 \ -I \ 0]$ and $[b_{22} \ 0 \ 0 ; 0 \ 0 \ I]$.</p>
<i>b11d</i>	<p>REAL for sbbcsd</p> <p>DOUBLE PRECISION for dbbcsd</p> <p>COMPLEX for cbbcsd</p> <p>DOUBLE COMPLEX for zbbcsd</p> <p>Array, size (<i>q</i>).</p>

When `?bbcsd` converges, `b11d` contains the cosines of $\theta(1), \dots, \theta(q)$. If `?bbcsd` fails to converge, `b11d` contains the diagonal of the partially reduced top left block.

`b11e`

REAL for `sbbcsd`

DOUBLE PRECISION for `dbbcsd`

COMPLEX for `cbbcsd`

DOUBLE COMPLEX for `zbbcsd`

Array, size $(q-1)$.

When `?bbcsd` converges, `b11e` contains zeros. If `?bbcsd` fails to converge, `b11e` contains the superdiagonal of the partially reduced top left block.

`b12d`

REAL for `sbbcsd`

DOUBLE PRECISION for `dbbcsd`

COMPLEX for `cbbcsd`

DOUBLE COMPLEX for `zbbcsd`

Array, size (q) .

When `?bbcsd` converges, `b12d` contains the negative sines of $\theta(1), \dots, \theta(q)$. If `?bbcsd` fails to converge, `b12d` contains the diagonal of the partially reduced top right block.

`b12e`

REAL for `sbbcsd`

DOUBLE PRECISION for `dbbcsd`

COMPLEX for `cbbcsd`

DOUBLE COMPLEX for `zbbcsd`

Array, size $(q-1)$.

When `?bbcsd` converges, `b12e` contains zeros. If `?bbcsd` fails to converge, `b12e` contains the superdiagonal of the partially reduced top right block.

`info`

INTEGER.

= 0: successful exit

< 0: if `info = -i`, the i -th argument has an illegal value

> 0: if `?bbcsd` did not converge, `info` specifies the number of nonzero entries in `phi`, and `b11d`, `b11e`, etc. contain the partially reduced matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?bbcsd` interface are as follows:

`theta` Holds the vector of length q .

`phi` Holds the vector of length $q-1$.

<i>u1</i>	Holds the matrix of size (p,p) .
<i>u2</i>	Holds the matrix of size $(m-p,m-p)$.
<i>v1t</i>	Holds the matrix of size (q,q) .
<i>v2t</i>	Holds the matrix of size $(m-q,m-q)$.
<i>b11d</i>	Holds the vector of length q .
<i>b11e</i>	Holds the vector of length $q-1$.
<i>b12d</i>	Holds the vector of length q .
<i>b12e</i>	Holds the vector of length $q-1$.
<i>b21d</i>	Holds the vector of length q .
<i>b21e</i>	Holds the vector of length $q-1$.
<i>b22d</i>	Holds the vector of length q .
<i>b22e</i>	Holds the vector of length $q-1$.
<i>jobsu1</i>	Indicates whether u_1 is computed. Must be 'Y' or 'O'.
<i>jobsu2</i>	Indicates whether u_2 is computed. Must be 'Y' or 'O'.
<i>jobv1t</i>	Indicates whether v_1^t is computed. Must be 'Y' or 'O'.
<i>jobv2t</i>	Indicates whether v_2^t is computed. Must be 'Y' or 'O'.
<i>trans</i>	Must be 'N' or 'T'.

See Also

[?orcsd/?uncsd](#)
[xerbla](#)

[?orbdb/?unbdb](#)

Simultaneously bidiagonalizes the blocks of a partitioned orthogonal/unitary matrix.

Syntax

```
call sorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call dorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call cunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call zunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

```
call orbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
```

```
call unbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routines `?orbdb/?unbdb` simultaneously bidiagonalizes the blocks of an m -by- m partitioned orthogonal matrix X :

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | & \\ & & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | & \\ & & q_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | & \\ & & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | & \\ & & q_2 \end{pmatrix}^H$$

x_{11} is p -by- q . q must not be larger than p , $m-p$, or $m-q$. Otherwise, x must be transposed and/or permuted in constant time using the `trans` and `signs` options.

The orthogonal/unitary matrices p_1 , p_2 , q_1 , and q_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. They are represented implicitly by Housholder vectors.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are q -by- q bidiagonal matrices represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q-1)$. b_{11} and b_{12} are upper bidiagonal, while b_{21} and b_{22} are lower bidiagonal. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] for details.

p_1 , p_2 , q_1 , and q_2 are represented as products of elementary reflectors. .

Input Parameters

<code>trans</code>	CHARACTER	
	= 'T':	x , u_1 , u_2 , v_1^t , v_2^t are stored in row-major order.
	otherwise	x , u_1 , u_2 , v_1^t , v_2^t are stored in column-major order.
<code>signs</code>	CHARACTER	
	= 'O':	The lower-left block is made nonpositive (the "other" convention).
	otherwise	The upper-right block is made nonpositive (the "default" convention).
<code>m</code>	INTEGER.	The number of rows and columns of the matrix X .

p	INTEGER. The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq \min(p, m-p, m-q)$.
x_{11}	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, size $(ldx11,*)$. On entry, the top-left block of the orthogonal/unitary matrix to be reduced.
$ldx11$	INTEGER. The leading dimension of the array X_{11} . If $trans = 'T'$, $ldx11 \geq p$. Otherwise, $ldx11 \geq q$.
x_{12}	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, size $(ldx12, m-q)$. On entry, the top-right block of the orthogonal/unitary matrix to be reduced.
$ldx12$	INTEGER. The leading dimension of the array X_{12} . If $trans = 'N'$, $ldx12 \geq p$. Otherwise, $ldx12 \geq m-q$.
x_{21}	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, size $(ldx21, q)$. On entry, the bottom-left block of the orthogonal/unitary matrix to be reduced.
$ldx21$	INTEGER. The leading dimension of the array X_{21} . If $trans = 'N'$, $ldx21 \geq m-p$. Otherwise, $ldx21 \geq q$.
x_{22}	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, size $(ldx22, m-q)$. On entry, the bottom-right block of the orthogonal/unitary matrix to be reduced.

<i>ldx22</i>	INTEGER. The leading dimension of the array X_{21} . If $trans = 'N'$, $ldx22 \geq m - p$. Otherwise, $ldx22 \geq m - q$.
<i>work</i>	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Workspace array, size (<i>lwork</i>).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. $lwork \geq m - q$ If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.

Output Parameters

<i>x11</i>	On exit, the form depends on <i>trans</i> : If $trans = 'N'$, the columns of the lower triangle of <i>x11</i> specify reflectors for p_1 , the rows of the upper triangle of $x11(1:q - 1, q:q - 1)$ specify reflectors for q_1 otherwise $trans = 'T'$, the rows of the upper triangle of <i>x11</i> specify reflectors for p_1 , the columns of the lower triangle of $x11(1:q - 1, q:q - 1)$ specify reflectors for q_1
<i>x12</i>	On exit, the form depends on <i>trans</i> : If $trans = 'N'$, the columns of the upper triangle of <i>x12</i> specify the first p reflectors for q_2 otherwise $trans = 'T'$, the columns of the lower triangle of <i>x12</i> specify the first p reflectors for q_2
<i>x21</i>	On exit, the form depends on <i>trans</i> : If $trans = 'N'$, the columns of the lower triangle of <i>x21</i> specify the reflectors for p_2 otherwise $trans = 'T'$, the columns of the upper triangle of <i>x21</i> specify the reflectors for p_2
<i>x22</i>	On exit, the form depends on <i>trans</i> : If $trans = 'N'$, the rows of the upper triangle of $x22(q+1:m-p, p+1:m-q)$ specify the last $m-p-q$ reflectors for q_2 otherwise $trans = 'T'$, the columns of the lower triangle of $x22(p+1:m-q, q+1:m-p)$ specify the last $m-p-q$ reflectors for p_2
<i>theta</i>	REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (q). The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles $theta$ and phi . See the Description section for details.

phi REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size ($q-1$). The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles $theta$ and phi . See the Description section for details.

taup1 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

tauq1 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size (q).

Scalar factors of the elementary reflectors that define q_1 .

tauq2 REAL for sorbdb

DOUBLE PRECISION for dorbdb

COMPLEX for cunbdb

DOUBLE COMPLEX for zunbdb

Array, size ($m-q$).

Scalar factors of the elementary reflectors that define q_2 .

info INTEGER.

= 0: successful exit

< 0: if *info* = *-i*, the *i*-th argument has an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?orbdb/?unbdb` interface are as follows:

<i>x11</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>q</i>).
<i>x12</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>m-q</i>).
<i>x21</i>	Holds the block of matrix <i>X</i> of size (<i>m-p</i> , <i>q</i>).
<i>x22</i>	Holds the block of matrix <i>X</i> of size (<i>m-p</i> , <i>m-q</i>).
<i>theta</i>	Holds the vector of length <i>q</i> .
<i>phi</i>	Holds the vector of length <i>q-1</i> .
<i>taup1</i>	Holds the vector of length <i>p</i> .
<i>taup2</i>	Holds the vector of length <i>m-p</i> .
<i>tauq1</i>	Holds the vector of length <i>q</i> .
<i>taupq2</i>	Holds the vector of length <i>m-q</i> .
<i>trans</i>	Must be 'N' or 'T'.
<i>signs</i>	Must be 'O' or 'D'.

See Also

[?orcscd/?uncscd](#)
[?orgqr](#)
[?ungqr](#)
[?orglq](#)
[?unglq](#)
[xerbla](#)

Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate [computational routines](#).

Driver routines are described in the following sections :

[Linear Least Squares \(LLS\) Problems](#)

[Generalized LLS Problems](#)

[Symmetric Eigenproblems](#)

[Nonsymmetric Eigenproblems](#)

[Singular Value Decomposition](#)

[Cosine-Sine Decomposition](#)

[Generalized Symmetric Definite Eigenproblems](#)

Generalized Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least squares problems. Table "Driver Routines for Solving LLS Problems" lists all such routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving LLS Problems

Routine Name	Operation performed
gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

[?gels](#)

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

```
call sgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call dgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call gels(a, b [,trans] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix A , or its transpose/ conjugate-transpose, using a QR or LQ factorization of A . It is assumed that A has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } \|b - A^*x\|_2$$

2. If $trans = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system $A^*X = B$.

3. If $trans = 'T'$ or $'C'$ and $m \geq n$: find the minimum norm solution of an undetermined system $A^H X = B$.

4. If $trans = 'T'$ or $'C'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } \|b - A^H x\|_2$$

Several right hand side vectors b and solution vectors x can be handled in a single call; they are formed by the columns of the right hand side matrix B and the solution matrix X (when coefficient matrix is A , B is m -by- $nrhs$ and X is n -by- $nrhs$; if the coefficient matrix is A^T or A^H , B is n -by- $nrhs$ and X is m -by- $nrhs$).

Input Parameters

<i>trans</i>	<p>CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>If <i>trans</i> = 'N', the linear system involves matrix A;</p> <p>If <i>trans</i> = 'T', the linear system involves the transposed matrix A^T (for real flavors only);</p> <p>If <i>trans</i> = 'C', the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).</p>
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	<p>INTEGER. The number of columns of the matrix A</p> <p>($n \geq 0$).</p>
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b, work</i>	<p>REAL for <i>sgels</i></p> <p>DOUBLE PRECISION for <i>dgels</i></p> <p>COMPLEX for <i>cgels</i></p> <p>DOUBLE COMPLEX for <i>zgels</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> contains the m-by-n matrix A.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>b(l db,*)</i> contains the matrix B of right hand side vectors.</p> <p>The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; must be at least $\min(m, n) + \max(1, m, n, nrhs)$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	On exit, overwritten by the factorization data as follows:
----------	--

if $m \geq n$, array a contains the details of the QR factorization of the matrix A as returned by `?geqrf`;

if $m < n$, array a contains the details of the LQ factorization of the matrix A as returned by `?gelqf`.

b If $info = 0$, b overwritten by the solution vectors, stored columnwise:

if $trans = 'N'$ and $m \geq n$, rows 1 to n of b contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $n+1$ to m in that column;

if $trans = 'N'$ and $m < n$, rows 1 to n of b contain the minimum norm solution vectors;

if $trans = 'T'$ or $'C'$ and $m \geq n$, rows 1 to m of b contain the minimum norm solution vectors;

if $trans = 'T'$ or $'C'$ and $m < n$, rows 1 to m of b contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $m+1$ to n in that column.

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of the triangular factor of A is zero, so that A does not have full rank; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gels` interface are the following:

a Holds the matrix A of size (m,n) .

b Holds the matrix of size $\max(m,n)$ -by- $nrhs$.
If $trans = 'N'$, then, on entry, the size of b is m -by- $nrhs$,
If $trans = 'T'$, then, on entry, the size of b is n -by- $nrhs$,

$trans$ Must be $'N'$ or $'T'$. The default value is $'N'$.

Application Notes

For better performance, try using $lwork = \min(m, n) + \max(1, m, n, nrhs) * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.

Syntax

```
call sgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call dgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call cgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
call zgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
call gelsy(a, b [,rank] [,jpvt] [,rcond] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?gelsy routine computes the minimum-norm solution to a real/complex linear least squares problem:

minimize $\|b - A*x\|_2$

using a complete orthogonal factorization of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient. Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*.

The routine first computes a *QR* factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix} Z$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than $1/rcond$. The order of R_{11} , *rank*, is the effective rank of *A*. Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^T \begin{pmatrix} T_{11}^{-1} Q_1^T B \\ 0 \end{pmatrix} \text{ for real flavors and}$$

$$X = PZ^H \begin{pmatrix} T_{11}^{-1} Q_1^H B \\ 0 \end{pmatrix} \text{ for complex flavors,}$$

where Q_1 consists of the first *rank* columns of Q .

The `?gelsy` routine is identical to the original deprecated `?gelsx` routine except for the following differences:

- The call to the subroutine `?geqpf` has been substituted by the call to the subroutine `?geqp3`, which is a BLAS-3 version of the QR factorization with column pivoting.
- The matrix B (the right hand side) is updated with BLAS-3.
- The permutation of the matrix B (the right hand side) is faster and more simple.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix A ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b, work</i>	REAL for <code>sgelsy</code> DOUBLE PRECISION for <code>dgelsy</code> COMPLEX for <code>cgelsy</code> DOUBLE COMPLEX for <code>zgelsy</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the m -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the m -by- <i>nrhs</i> right hand side matrix B . The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>jpvt</i>	INTEGER. Array, size at least $\max(1, n)$. On entry, if <i>jpvt</i> (<i>i</i>) $\neq 0$, the <i>i</i> -th column of A is permuted to the front of AP , otherwise the <i>i</i> -th column of A is a free column.

<i>rcond</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>rcond</i> is used to determine the effective rank of <i>A</i> , which is defined as the order of the largest leading triangular submatrix R_{11} in the <i>QR</i> factorization with pivoting of <i>A</i> , whose estimated condition number $< 1/rcond$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for <code>cgelsy</code> DOUBLE PRECISION for <code>zgelsy</code> . Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>a</i>	On exit, overwritten by the details of the complete orthogonal factorization of <i>A</i> .
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>jpvt</i>	On exit, if <i>jpvt</i> (<i>i</i>) = <i>k</i> , then the <i>i</i> -th column of <i>AP</i> was the <i>k</i> -th column of <i>A</i> .
<i>rank</i>	INTEGER. The effective rank of <i>A</i> , that is, the order of the submatrix R_{11} . This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gelsy` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- <i>nrhs</i> . On entry, contains the <i>m</i> -by- <i>nrhs</i> right hand side matrix <i>B</i> , On exit, overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>jpvt</i>	Holds the vector of length <i>n</i> . Default value for this element is <i>jpvt</i> (<i>i</i>) = 0.
<i>rcond</i>	Default value for this element is <i>rcond</i> = 100*EPSILON(1.0_wp).

Application Notes

For real flavors:

The unblocked strategy requires that:

$lwork \geq \max(mn+3n+1, 2*mn + nrhs)$,

where $mn = \min(m, n)$.

The block algorithm requires that:

$lwork \geq \max(mn+2n+nb*(n+1), 2*mn+nb*nrhs)$,

where nb is an upper bound on the blocksize returned by [ilaenv](#) for the routines [sgeqp3/dgeqp3](#), [stzrzf/dtzrzf](#), [stzrqf/dtzrqf](#), [sormqr/dormqr](#), and [sormrz/dormrz](#).

For complex flavors:

The unblocked strategy requires that:

$lwork \geq mn + \max(2*mn, n+1, mn + nrhs)$,

where $mn = \min(m, n)$.

The block algorithm requires that:

$lwork < mn + \max(2*mn, nb*(n+1), mn+mn*nb, mn+ nb*nrhs)$,

where nb is an upper bound on the blocksize returned by [ilaenv](#) for the routines [cgeqp3/zgeqp3](#), [ctzrzf/ztzrzf](#), [ctzrqf/ztzrqf](#), [cunmqr/zunmqr](#), and [cunmrz/zunmrz](#).

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

```
call sgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call dgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call cgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
call zgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
call gelss(a, b [,rank] [,s] [,rcond] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes the minimum norm solution to a real linear least squares problem:

minimize $\|b - A*x\|_2$

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X . The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
$a, b, work$	REAL for <code>sgelss</code> DOUBLE PRECISION for <code>dgelss</code> COMPLEX for <code>cgelss</code> DOUBLE COMPLEX for <code>zgelss</code> . Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $b(l db,*)$ contains the m -by- $nrhs$ right hand side matrix B . The second dimension of b must be at least $\max(1, nrhs)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldb	INTEGER. The leading dimension of b ; must be at least $\max(1, m, n)$.
$rcond$	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. $rcond$ is used to determine the effective rank of A . Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond < 0$, machine precision is used instead.
$lwork$	INTEGER. The size of the $work$ array; $lwork \geq 1$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.
$rwork$	REAL for <code>cgelss</code> DOUBLE PRECISION for <code>zgelss</code> .

Workspace array used in complex flavors only. size at least $\max(1, 5*\min(m, n))$.

Output Parameters

<i>a</i>	On exit, the first $\min(m, n)$ rows of <i>a</i> are overwritten with the matrix of right singular vectors of <i>A</i> , stored row-wise.
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> . If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the <i>i</i> -th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, \min(m, n))$. The singular values of <i>A</i> in decreasing order. The condition number of <i>A</i> in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.
<i>rank</i>	INTEGER. The effective rank of <i>A</i> , that is, the number of singular values which are greater than $rcond * s(1)$.
<i>work(1)</i>	If $info = 0$, on exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$, then the algorithm for computing the SVD failed to converge; <i>i</i> indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gelss` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m,n) .
<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- <i>nrhs</i> . On entry, contains the <i>m</i> -by- <i>nrhs</i> right hand side matrix <i>B</i> , On exit, overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>s</i>	Holds the vector of length $\min(m,n)$.
<i>rcond</i>	Default value for this element is $rcond = 100*EPSILON(1.0_WP)$.

Application Notes

For real flavors:

$lwork \geq 3*\min(m, n) + \max(2*\min(m, n), \max(m, n), nrhs)$

For complex flavors:

$$lwork \geq 2 * \min(m, n) + \max(m, n, nrhs)$$

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

sgelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

```
call sgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call cgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
call zgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork, info)
call gelsd(a, b [,rank] [,s] [,rcond] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the minimum-norm solution to a real linear least squares problem:

$$\text{minimize } \|b - A*x\|_2$$

using the singular value decomposition (SVD) of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient.

Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*.

The problem is solved in three steps:

1. Reduce the coefficient matrix *A* to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of *A* is determined by treating as zero those singular values which are less than *rcond* times the largest singular value.

The routine uses auxiliary routines [lals0](#) and [lalsa](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>a, b, work</i>	<p>REAL for <i>sgelsd</i></p> <p>DOUBLE PRECISION for <i>dgelsd</i></p> <p>COMPLEX for <i>cgelsd</i></p> <p>DOUBLE COMPLEX for <i>zgelsd</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>b(ldb,*)</i> contains the <i>m</i>-by-<i>nrhs</i> right hand side matrix <i>B</i>. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>rcond</i>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p><i>rcond</i> is used to determine the effective rank of <i>A</i>. Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond \leq 0$, machine precision is used instead.</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; $lwork \geq 1$.</p> <p>If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the array <i>work</i> and the minimum sizes of the arrays <i>rwork</i> and <i>iwork</i>, and returns these values as the first entries of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>
<i>iwork</i>	INTEGER. Workspace array. See <i>Application Notes</i> for the suggested dimension of <i>iwork</i> .
<i>rwork</i>	<p>REAL for <i>cgelsd</i></p> <p>DOUBLE PRECISION for <i>zgelsd</i>.</p> <p>Workspace array, used in complex flavors only. See <i>Application Notes</i> for the suggested dimension of <i>rwork</i>.</p>

Output Parameters

<i>a</i>	On exit, <i>A</i> has been overwritten.
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> . If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the <i>i</i> -th column is given by the sum of squares of modulus of elements <i>n</i> +1: <i>m</i> in that column.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, \min(m, n))$. The singular values of <i>A</i> in decreasing order. The condition number of <i>A</i> in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.
<i>rank</i>	INTEGER. The effective rank of <i>A</i> , that is, the number of singular values which are greater than $rcond * s(1)$.
<i>work</i> (1)	If <i>info</i> = 0, on exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>rwork</i> (1)	If <i>info</i> = 0, on exit, <i>rwork</i> (1) returns the minimum size of the workspace array <i>iwork</i> required for optimum performance.
<i>iwork</i> (1)	If <i>info</i> = 0, on exit, <i>iwork</i> (1) returns the minimum size of the workspace array <i>iwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm for computing the SVD failed to converge; <i>i</i> indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gelsd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- <i>nrhs</i> . On entry, contains the <i>m</i> -by- <i>nrhs</i> right hand side matrix <i>B</i> , On exit, overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>s</i>	Holds the vector of length $\min(m,n)$.
<i>rcond</i>	Default value for this element is $rcond = 100 * EPSILON(1.0_WP)$.

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on m , n and $nrhs$. The size $lwork$ of the workspace array $work$ must be as given below.

For real flavors:

If $m \geq n$,

$$lwork \geq 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2;$$

If $m < n$,

$$lwork \geq 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2;$$

For complex flavors:

If $m \geq n$,

$$lwork < 2n + n*nrhs;$$

If $m < n$,

$$lwork \geq 2m + m*nrhs;$$

where $smlsiz$ is returned by `ilaenv` and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and

$$nlvl = \text{INT}(\log_2(\min(m, n)/(smlsiz+1))) + 1.$$

For good performance, $lwork$ should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The dimension of the workspace array $iwork$ must be at least

$$3*\min(m, n)*nlvl + 11*\min(m, n).$$

The dimension of the workspace array $iwork$ (for complex flavors) must be at least $\max(1, lrwork)$.

$$lrwork \geq 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2 \text{ if } m \geq n, \text{ and}$$

$$lrwork \geq 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2 \text{ if } m < n.$$

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least squares problems. [Table "Driver Routines for Solving Generalized LLS Problems"](#) lists all such routines. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

[?gglse](#)

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

```
call sgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call dgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call cgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call zgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call gglsse(a, b, c, d, x [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine solves the linear equality-constrained least squares (LSE) problem:

minimize $\|c - A*x\|^2$ subject to $B*x = d$

where A is an m -by- n matrix, B is a p -by- n matrix, c is a given m -vector, and d is a given p -vector. It is assumed that $p \leq n \leq m+p$, and

$$B = \begin{pmatrix} 0 & R \end{pmatrix} * Q, \quad A = Z * T * Q$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized RQ factorization of the matrices (B, A) given by

$$B = \begin{pmatrix} 0 & R \end{pmatrix} * Q, \quad A = Z * T * Q$$

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
p	INTEGER. The number of rows of the matrix B ($0 \leq p \leq n \leq m+p$).
$a, b, c, d, work$	REAL for sgglsse

DOUBLE PRECISION for `dggls`

COMPLEX for `cggls`

DOUBLE COMPLEX for `zggls`.

Arrays:

$a(lda,*)$ contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$b(ldb,*)$ contains the p -by- n matrix B .

The second dimension of b must be at least $\max(1, n)$.

$c(*)$, size at least $\max(1, m)$, contains the right hand side vector for the least squares part of the LSE problem.

$d(*),,$ size at least $\max(1, p)$, contains the right hand side vector for the constrained equation.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, p)$.

$lwork$ INTEGER. The size of the $work$ array;

$lwork \geq \max(1, m+n+p)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a The elements on and above the diagonal contain the $\min(m, n)$ -by- n upper trapezoidal matrix T as returned by `?ggrqf`.

x REAL for `sggls`
The solution of the LSE problem.

b On exit, the upper right triangle of the subarray $b(1:p, n-p+1:n)$ contains the p -by- p upper triangular matrix R as returned by `?ggrqf`.

d On exit, d is destroyed.

c On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements $n-p+1$ to m of vector c .

$work(1)$ If $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the upper triangular factor R associated with B in the generalized RQ factorization of the pair (B, A) is singular, so that $\text{rank}(B) < p$; the least squares solution could not be computed.

If $info = 2$, the $(n-p)$ -by- $(n-p)$ part of the upper trapezoidal factor T associated with A in the generalized RQ factorization of the pair (B, A) is singular, so that

$$\text{rank} \begin{pmatrix} A \\ B \end{pmatrix} < n$$

; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gglsse` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>b</i>	Holds the matrix B of size (p,n) .
<i>c</i>	Holds the vector of length (m) .
<i>d</i>	Holds the vector of length (p) .
<i>x</i>	Holds the vector of length n .

Application Notes

For optimum performance, use

$$lwork \geq p + \min(m, n) + \max(m, n) * nb,$$

where nb is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

```
call sggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call dggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call cggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call zggglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call ggglm(a, b, d, x, y [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine solves a general Gauss-Markov linear model (GLM) problem:

minimize_x $\|y\|_2$ subject to $d = A^*x + B^*y$

where A is an n -by- m matrix, B is an n -by- p matrix, and d is a given n -vector. It is assumed that $m \leq n \leq m+p$, and $\text{rank}(A) = m$ and $\text{rank}(AB) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y , which is obtained using a generalized QR factorization of the matrices (A, B) given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \quad B = Q^* T^* Z.$$

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

minimize_x $\|B^{-1}(d - A^*x)\|_2$.

Input Parameters

n INTEGER. The number of rows of the matrices A and B ($n \geq 0$).

m INTEGER. The number of columns in A ($m \geq 0$).

p INTEGER. The number of columns in B ($p \geq n - m$).

$a, b, d, work$ REAL for `sggglm`
DOUBLE PRECISION for `dggglm`
COMPLEX for `cggglm`
DOUBLE COMPLEX for `zggglm`.

Arrays:

$a(lda,*)$ contains the n -by- m matrix A .

The second dimension of a must be at least $\max(1, m)$.

$b(l db,*)$ contains the n -by- p matrix B .

The second dimension of b must be at least $\max(1, p)$.

$d(*)$, size at least $\max(1, n)$, contains the left hand side of the GLM equation.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, n)$.

$lwork$ INTEGER. The size of the $work$ array; $lwork \geq \max(1, n+m+p)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>x, y</i>	REAL for sggglm DOUBLE PRECISION for dggglm COMPLEX for cggglm DOUBLE COMPLEX for zggglm. Arrays $x(*)$, $y(*)$. size at least $\max(1, m)$ for x and at least $\max(1, p)$ for y . On exit, x and y are the solutions of the GLM problem.
<i>a</i>	On exit, the upper triangular part of the array a contains the m -by- m upper triangular matrix R .
<i>b</i>	On exit, if $n \leq p$, the upper right triangle of the subarray $b(1:n, p-n+1:p)$ contains the n -by- n upper triangular matrix T as returned by ?ggrqf ; if $n > p$, the elements on and above the $(n-p)$ -th subdiagonal contain the n -by- p upper trapezoidal matrix T .
<i>d</i>	On exit, d is destroyed
<i>work(1)</i>	If $info = 0$, on exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, the upper triangular factor R associated with A in the generalized QR factorization of the pair (A, B) is singular, so that $\text{rank}(A) < m$; the least squares solution could not be computed. If $info = 2$, the bottom $(n-m)$ -by- $(n-m)$ part of the upper trapezoidal factor T associated with B in the generalized QR factorization of the pair (A, B) is singular, so that $\text{rank}(AB) < n$; the least squares solution could not be computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine [ggglm](#) interface are the following:

<i>a</i>	Holds the matrix A of size (n, m) .
<i>b</i>	Holds the matrix B of size (n, p) .

d	Holds the vector of length n .
x	Holds the vector of length (m).
y	Holds the vector of length (p).

Application Notes

For optimum performance, use

$$lwork \geq m + \min(n, p) + \max(n, p) * nb,$$

where nb is an upper bound for the optimal blocksizes for `?geqrf`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrq`.

You may set $lwork$ to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Symmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
syeval / heeval	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
syevald / heevald	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
syevalx / heevalx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
syevalr / heevalr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
speval / hpeval	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
spevald / hpevald	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
spevalx / hpevalx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
sbeval / hbeval	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
sbevald / hbevald	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
sbevalx / hbevalx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
steval	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Routine Name	Operation performed
stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

[?syev](#)

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

```
call ssyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call dsyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call syev(a, w [,jobz] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A .

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

<code>jobz</code>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <code>jobz = 'N'</code>, then only eigenvalues are computed.</p> <p>If <code>jobz = 'V'</code>, then eigenvalues and eigenvectors are computed.</p>
<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <code>uplo = 'U'</code>, <code>a</code> stores the upper triangular part of A.</p> <p>If <code>uplo = 'L'</code>, <code>a</code> stores the lower triangular part of A.</p>
<code>n</code>	<p>INTEGER. The order of the matrix A ($n \geq 0$).</p>
<code>a, work</code>	<p>REAL for <code>ssyev</code></p> <p>DOUBLE PRECISION for <code>dsyev</code></p> <p><code>a(lda,*)</code> is an array containing either upper or lower triangular part of the symmetric matrix A, as specified by <code>uplo</code>.</p> <p>The second dimension of <code>a</code> must be at least $\max(1, n)$.</p> <p><code>work</code> is a workspace array, its dimension $\max(1, lwork)$.</p>
<code>lda</code>	<p>INTEGER. The leading dimension of the array <code>a</code>.</p>

Must be at least $\max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraint: $lwork \geq \max(1, 3n-1)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

On exit, if $jobz = 'V'$, then if $info = 0$, array *a* contains the orthonormal eigenvectors of the matrix *A*.

If $jobz = 'N'$, then on exit the lower triangle

(if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of *A*, including the diagonal, is overwritten.

w

REAL for *ssyev*

DOUBLE PRECISION for *dsyev*

Array, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order.

work(1)

On exit, if $lwork > 0$, then *work(1)* returns the required minimal size of *work*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the *i*-th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *syev* interface are the following:

a

Holds the matrix *A* of size (n, n) .

w

Holds the vector of length *n*.

job

Must be 'N' or 'V'. The default value is 'N'.

uplo

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance set $lwork \geq (nb+2) * n$, where nb is the blocksize for `?sytrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

If $lwork$ has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array $work$. This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call zheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call heev(a, w [,jobz] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A .

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be `heevr` function as its underlying algorithm is faster and uses less workspace.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
 If *jobz* = 'N', then only eigenvalues are computed.
 If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	<p>COMPLEX for cheev</p> <p>DOUBLE COMPLEX for zheev</p> <p>Arrays:</p> <p><i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>. C</p> <p>constraint: $lwork \geq \max(1, 2n-1)$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>
<i>rwork</i>	<p>REAL for cheev</p> <p>DOUBLE PRECISION for zheev.</p> <p>Workspace array, size at least $\max(1, 3n-2)$.</p>
Output Parameters	
<i>a</i>	<p>On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix <i>A</i>.</p> <p>If <i>jobz</i> = 'N', then on exit the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i>, including the diagonal, is overwritten.</p>
<i>w</i>	<p>REAL for cheev</p> <p>DOUBLE PRECISION for zheev</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.</p>
<i>work(1)</i>	On exit, if <i>lwork</i> > 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `heev` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>job</code>	Must be 'N' or 'V'. The default value is 'N'.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use

$$lwork \geq (nb+1)*n,$$

where nb is the blocksize for `?hetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

```
call ssyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

```
call dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

```
call syevd(a, w [,jobz] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A . In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be `syevr` function as its underlying algorithm is faster and uses less workspace. `?syevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a</i>	REAL for <code>ssyevd</code> DOUBLE PRECISION for <code>dsyevd</code> Array, size (<i>lda</i> , *). <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>work</i>	REAL for <code>ssyevd</code> DOUBLE PRECISION for <code>dsyevd</code> . Workspace array, size at least <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if $n \leq 1$, then $lwork \geq 1$; if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq 2*n + 1$; if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq 2*n^2 + 6*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.
Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array $iwork$.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w REAL for `ssyevd`
DOUBLE PRECISION for `dsyevd`
Array, size at least $\max(1, n)$.
If $info = 0$, contains the eigenvalues of the matrix A in ascending order. See also *info*.

a If $jobz = 'V'$, then on exit this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A .

work(1) On exit, if $lwork > 0$, then *work*(1) returns the required minimal size of $lwork$.

iwork(1) On exit, if $liwork > 0$, then *iwork*(1) returns the required minimal size of $liwork$.

info INTEGER.
If $info = 0$, the execution is successful.
If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.
If $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.
If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `syevd` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>jobz</code>	Must be 'N' or 'V'. The default value is 'N'.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run, or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [heevd](#)

?heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

```
call cheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
call zheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
call heevd(a, w [,job] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A . In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A^*z_i = \lambda_i^*z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace. `?heevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<p>COMPLEX for <code>cheevd</code></p> <p>DOUBLE COMPLEX for <code>zheevd</code></p> <p>Array, size (<i>lda</i>, *).</p> <p><i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>work</i>	<p>COMPLEX for <code>cheevd</code></p> <p>DOUBLE COMPLEX for <code>zheevd</code>.</p> <p>Workspace array, size $\max(1, lwork)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>. Constraints:</p> <p>if $n \leq 1$, then $lwork \geq 1$;</p> <p>if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq n+1$;</p> <p>if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq n^2+2*n$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>lrwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.</p>
<i>rwork</i>	<p>REAL for <code>cheevd</code></p> <p>DOUBLE PRECISION for <code>zheevd</code></p>

Workspace array, size at least $lrwork$.

$lrwork$

INTEGER.

The dimension of the array $rwork$. Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $job = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $job = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$. Constraints: if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w

REAL for `cheevd`

DOUBLE PRECISION for `zheevd`

Array, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order. See also $info$.

a

If $jobz = 'V'$, then on exit this array is overwritten by the unitary matrix Z which contains the eigenvectors of A .

$work(1)$

On exit, if $lwork > 0$, then the real part of $work(1)$ returns the required minimal size of $lwork$.

$rwork(1)$

On exit, if $lrwork > 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.

$iwork(1)$

On exit, if $liwork > 0$, then $iwork(1)$ returns the required minimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $mod(info, n+1)$.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `heevd` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>w</i>	Holds the vector of length (n) .
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1, lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1, lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [syevd](#). See also [hpevd](#) for matrices held in packed storage, and [hbevd](#) for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
call ssyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, iwork, ifail, info)
```

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, iwork, ifail, info)
```

```
call syevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be `syevr` function as its underlying algorithm is faster and uses less workspace. `?syevx` is faster for a few selected eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval $(vl, vu]$ will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a, work</i>	REAL for <code>ssyevx</code> DOUBLE PRECISION for <code>dsyevx</code> . Arrays: <i>a(lda,*)</i> is an array containing either upper or lower triangular part of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>vl, vu</i>	REAL for <code>ssyevx</code> DOUBLE PRECISION for <code>dsyevx</code> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.

<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned.</p> <p>Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$.</p> <p>Not referenced if <i>range</i> = 'A' or 'V'.</p>
<i>abstol</i>	<p>REAL for <i>ssyevx</i></p> <p>DOUBLE PRECISION for <i>dsyevx</i>.</p> <p>The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$.</p> <p>If <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>If $n \leq 1$ then $lwork \geq 1$, otherwise $lwork = 8 * n$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>
<i>iwork</i>	<p>INTEGER. Workspace array, size at least $\max(1, 5n)$.</p>
Output Parameters	
<i>a</i>	<p>On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i>, including the diagonal, is overwritten.</p>
<i>m</i>	<p>INTEGER. The total number of eigenvalues found;</p> <p>$0 \leq m \leq n$.</p> <p>If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.</p>
<i>w</i>	<p>REAL for <i>ssyevx</i></p> <p>DOUBLE PRECISION for <i>dsyevx</i></p> <p>Array, size at least $\max(1, n)$. The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.</p>
<i>z</i>	<p>REAL for <i>ssyevx</i></p> <p>DOUBLE PRECISION for <i>dsyevx</i>.</p> <p>Array <i>z</i>(<i>ldz</i>,*) contains eigenvectors.</p> <p>The second dimension of <i>z</i> must be at least $\max(1, m)$.</p>

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$ifail$

INTEGER.

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, then $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'V'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `syevx` interface are the following:

a	Holds the matrix A of size (n, n) .
w	Holds the vector of length n .
a	Holds the matrix A of size (m, n) .
$ifail$	Holds the vector of length n .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
vl	Default value for this element is $vl = -\text{HUGE}(vl)$.
vu	Default value for this element is $vu = \text{HUGE}(vu)$.
il	Default value for this argument is $il = 1$.

<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use $lwork \geq (nb+3)*n$, where *nb* is the maximum of the blocksize for `?sytrd` and `?ormtr` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If *lwork* has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array *work*. This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||$ is used as tolerance, where $||T||$ is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * \lambda_{\text{mach}}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \lambda_{\text{mach}}('S')$.

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call cheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)
```

```
call zheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)
```

```
call heevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermetian eigenvalue problems the default choice should be `heevr` function as its underlying algorithm is faster and uses less workspace. `?heevx` is faster for a few selected eigenvalues.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A', 'V', or 'I'.</p> <p>If <i>range</i> = 'A', all eigenvalues will be found.</p> <p>If <i>range</i> = 'V', all eigenvalues in the half-open interval (<i>vl</i>, <i>vu</i>] will be found.</p> <p>If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A ($n \geq 0$).</p>
<i>a, work</i>	<p>COMPLEX for <code>cheevx</code></p> <p>DOUBLE COMPLEX for <code>zheevx</code>.</p> <p>Arrays:</p> <p><i>a</i>(<i>lda</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix A, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. Must be at least $\max(1, n)$.</p>
<i>vl, vu</i>	<p>REAL for <code>cheevx</code></p> <p>DOUBLE PRECISION for <code>zheevx</code>.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints:</p> <p>$1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if <i>range</i> = 'A' or 'V'.</p>

<i>abstol</i>	<p>REAL for <code>cheevx</code></p> <p>DOUBLE PRECISION for <code>zheevx</code>. The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$.</p> <p>If <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>$lwork \geq 1$ if $n \leq 1$; otherwise at least $2 * n$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>
<i>rwork</i>	<p>REAL for <code>cheevx</code></p> <p>DOUBLE PRECISION for <code>zheevx</code>.</p> <p>Workspace array, size at least $\max(1, 7n)$.</p>
<i>iwork</i>	<p>INTEGER. Workspace array, size at least $\max(1, 5n)$.</p>

Output Parameters

<i>a</i>	<p>On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i>, including the diagonal, is overwritten.</p>
<i>m</i>	<p>INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.</p> <p>If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.</p>
<i>w</i>	<p>REAL for <code>cheevx</code></p> <p>DOUBLE PRECISION for <code>zheevx</code></p> <p>Array, size $\max(1, n)$. The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.</p>
<i>z</i>	<p>COMPLEX for <code>cheevx</code></p> <p>DOUBLE COMPLEX for <code>zheevx</code>.</p> <p>Array <i>z</i>(<i>ldz</i>,*) contains eigenvectors.</p> <p>The second dimension of <i>z</i> must be at least $\max(1, m)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>).</p> <p>If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p>

If $jobz = 'N'$, then z is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

<i>work(1)</i>	On exit, if $lwork > 0$, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>ifail</i>	INTEGER. Array, size at least $\max(1, n)$. If $jobz = 'V'$, then if $info = 0$, the first m elements of <i>ifail</i> are zero; if $info > 0$, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If $jobz = 'V'$, then <i>ifail</i> is not referenced.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array <i>ifail</i> .

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `heevx` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>z</i>	Holds the matrix Z of size (n, n) .
<i>ifail</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_wp$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> is present and z is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows: *range* = 'V', if one of or both *vl* and *vu* are present, *range* = 'I', if one of or both *il* and *iu* are present, *range* = 'A', if none of *vl*, *vu*, *il*, *iu* is present, Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where *nb* is the maximum of the blocksize for `?hetrd` and `?unmtr` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||$ will be used in its place, where $||T||$ is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \lambda_{\text{lamch}}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \lambda_{\text{lamch}}('S')$.

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

```
call ssyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)
```

```
call dsyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)
```

```
call syevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix *A* to tridiagonal form *T*. Then, whenever possible, `?syevr` calls `stemr` to compute the eigenspectrum using Relatively Robust Representations. `stemr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L * D * L^T$ representations

(also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of T :

- a.** Compute $T - \sigma^* I = L^* D^* L^{*T}$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?syevr` calls `stemr` when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. `?syevr` calls `stebz` and `stein` on non-IEEE machines and when partial spectrum requests are made.

Note that `?syevr` is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> . For <i>range</i> = 'V' or 'I' and $iu - il < n - 1$, <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for <code>ssyevr</code> DOUBLE PRECISION for <code>dsyevr</code> . Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

vl, vu REAL for `ssyevr`

DOUBLE PRECISION for `dsyevr`.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or $'I'$, vl and vu are not referenced.

il, iu INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint:

$1 \leq il \leq iu \leq n$, if $n > 0$;

$il=1$ and $iu=0$, if $n = 0$.

If $range = 'A'$ or $'V'$, il and iu are not referenced.

$abstol$ REAL for `ssyevr`

DOUBLE PRECISION for `dsyevr`. The absolute error tolerance to which each eigenvalue/eigenvector is required.

If $jobz = 'V'$, the eigenvalues and eigenvectors output have residual norms bounded by $abstol$, and the dot products between different eigenvectors are bounded by $abstol$.

If $abstol < n * \text{eps} * ||T||$, then $n * \text{eps} * ||T||$ is used instead, where eps is the machine precision, and $||T||$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\text{eps} * ||T||$ irrespective of $abstol$.

If high relative accuracy is important, set $abstol$ to `?lamch('S')`.

ldz INTEGER. The leading dimension of the output array z .

Constraints:

$ldz \geq 1$ if $jobz = 'N'$ and

$ldz \geq \max(1, n)$ if $jobz = 'V'$.

$lwork$ INTEGER.

The dimension of the array $work$.

Constraint: $lwork \geq \max(1, 26n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*, $lwork \geq \max(1, 10n)$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [xerbla](#).

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V' the exact value of *m* is not known in advance.

w, *z* REAL for *ssyevr*

DOUBLE PRECISION for *dsyevr*.

Arrays:

w(*), size at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*);

z(*ldz*,*), the second dimension of *z* must be at least $\max(1, m)$.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If *jobz* = 'N', then *z* is not referenced. Note that you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

isuppz INTEGER.

Array, size at least $2 * \max(1, m)$.

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i*-1) through *isuppz*(2*i*). Referenced only if eigenvectors are needed (*jobz* = 'V') and all eigenvalues are needed, that is, *range* = 'A' or *range* = 'I' and *il* = 1 and *iu* = *n*.

work(1) On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, an internal error has occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `syevr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length (2* <i>m</i>), where the values (2* <i>m</i>) are significant.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vu</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted. Note that there will be an error condition if <i>isuppz</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use $lwork \geq (nb+6) * n$, where *nb* is the maximum of the blocksize for `?sytrd` and `?ormtr` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work, iwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

```
call cheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call heevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T with a call to [hetrd](#). Then, whenever possible, `?heevr` calls [stegr](#) to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of T :

- Compute $T - \sigma^*I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?heevr` calls [stemr](#) when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or [stebz](#) and [stein](#) on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `?heevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $v_l < \lambda(i) \leq v_u$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p> <p>For <i>range</i> = 'V' or 'I', <i>sstebz/dstebz</i> and <i>cstein/zstein</i> are called.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>a, work</i>	<p>COMPLEX for <i>cheevr</i></p> <p>DOUBLE COMPLEX for <i>zheevr</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>.</p> <p>Must be at least $\max(1, n)$.</p>
<i>vl, vu</i>	<p>REAL for <i>cheevr</i></p> <p>DOUBLE PRECISION for <i>zheevr</i>.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $v_l < v_u$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>REAL for <i>cheevr</i></p> <p>DOUBLE PRECISION for <i>zheevr</i>.</p>

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If $jobz = 'V'$, the eigenvalues and eigenvectors output have residual norms bounded by $abstol$, and the dot products between different eigenvectors are bounded by $abstol$.

If $abstol < n * \epsilon * ||T||$, then $n * \epsilon * ||T||$ is used instead, where ϵ is the machine precision, and $||T||$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\epsilon * ||T||$ irrespective of $abstol$.

If high relative accuracy is important, set $abstol$ to $?lamch('S')$.

ldz

INTEGER. The leading dimension of the output array *z*. Constraints:

$ldz \geq 1$ if $jobz = 'N'$;

$ldz \geq \max(1, n)$ if $jobz = 'V'$.

lwork

INTEGER.

The dimension of the array *work*.

Constraint: $lwork \geq \max(1, 2n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for *cheevr*

DOUBLE PRECISION for *zheevr*.

Workspace array, size $\max(1, lwork)$.

lrwork

INTEGER.

The dimension of the array *rwork*;

$lrwork \geq \max(1, 24n)$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

iwork

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*,

$liwork \geq \max(1, 10n)$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	<p>INTEGER. The total number of eigenvalues found,</p> <p>$0 \leq m \leq n$.</p> <p>If <i>range</i> = 'A', <i>m</i> = <i>n</i>, if <i>range</i> = 'I', <i>m</i> = <i>iu-il+1</i>, and if <i>range</i> = 'V' the exact value of <i>m</i> is not known in advance.</p>
<i>w</i>	<p>REAL for <i>cheevr</i></p> <p>DOUBLE PRECISION for <i>zheevr</i>.</p> <p>Array, size at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in <i>w</i>(1) to <i>w</i>(<i>m</i>).</p>
<i>z</i>	<p>COMPLEX for <i>cheevr</i></p> <p>DOUBLE COMPLEX for <i>zheevr</i>.</p> <p>Array <i>z</i>(<i>ldz</i>,*), the second dimension of <i>z</i> must be at least $\max(1, m)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>).</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p> <p>Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.</p>
<i>isuppz</i>	<p>INTEGER.</p> <p>Array, size at least $2 * \max(1, m)$.</p> <p>The support of the eigenvectors in <i>z</i>, i.e., the indices indicating the nonzero elements in <i>z</i>. The <i>i</i>-th eigenvector is nonzero only in elements <i>isuppz</i>(2<i>i</i>-1) through <i>isuppz</i>(2<i>i</i>). Referenced only if eigenvectors are needed (<i>jobz</i> = 'V') and all eigenvalues are needed, that is, <i>range</i> = 'A' or <i>range</i> = 'I' and <i>il</i> = 1 and <i>iu</i> = <i>n</i>.</p>
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if <i>info</i> = 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, an internal error has occurred.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `heevr` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>z</code>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<code>isuppz</code>	Holds the vector of length $(2*n)$, where the values $(2*m)$ are significant.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vl</code>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<code>vu</code>	Default value for this element is $vu = \text{HUGE}(vu)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: $jobz = 'V'$, if <code>z</code> is present, $jobz = 'N'$, if <code>z</code> is omitted. Note that there will be an error condition if <code>isuppz</code> is present and <code>z</code> is omitted.
<code>range</code>	Restored based on the presence of arguments <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> as follows: $range = 'V'$, if one of or both <code>vl</code> and <code>vu</code> are present, $range = 'I'$, if one of or both <code>il</code> and <code>iu</code> are present, $range = 'A'$, if none of <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> is present. Note that there will be an error condition if one of or both <code>vl</code> and <code>vu</code> are present and at the same time one of or both <code>il</code> and <code>iu</code> are present.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where nb is the maximum of the blocksize for `?hetrd` and `?unmtr` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of `lwork` (or `lrwork`, or `liwork`) for the first run or set $lwork = -1$ ($lrwork = -1$, $liwork = -1$).

If you choose the first option and set any of admissible `lwork` (or `lrwork`, `liwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `rwork`, `iwork`) on exit. Use this value (`work(1)`, `rwork(1)`, `iwork(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `rwork`, `iwork`). This operation is called a workspace query.

Note that if you set `lwork` (`lrwork`, `liwork`) to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of `?stemr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see `?stemr` and these references:

- Inderjit S. Dhillon and Beresford N. Parlett: "Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices," *Linear Algebra and its Applications*, 387(1), pp. 1-28, August 2004.
- Inderjit Dhillon and Beresford Parlett: "Orthogonal Eigenvectors and Relative Gaps," *SIAM Journal on Matrix Analysis and Applications*, Vol. 25, 2004. Also LAPACK Working Note 154.
- Inderjit Dhillon: "A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

`?spev`

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call spev(ap, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage.

Input Parameters

<code>jobz</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>job = 'N'</code> , then only eigenvalues are computed. If <code>job = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>ap</code> stores the packed upper triangular part of A . If <code>uplo = 'L'</code> , <code>ap</code> stores the packed lower triangular part of A .
<code>n</code>	INTEGER. The order of the matrix A ($n \geq 0$).
<code>ap, work</code>	REAL for <code>sspev</code> DOUBLE PRECISION for <code>dspev</code> Arrays: Array <code>ap(*)</code> contains the packed upper or lower triangle of symmetric matrix A , as specified by <code>uplo</code> . The size of <code>ap</code> must be at least $\max(1, n*(n+1)/2)$. <code>work(*)</code> is a workspace array, size at least $\max(1, 3n)$.

ldz INTEGER. The leading dimension of the output array *z*. Constraints:
 if *jobz* = 'N', then $ldz \geq 1$;
 if *jobz* = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, z REAL for *sspev*
 DOUBLE PRECISION for *dspev*

Arrays:
w(*), size at least $\max(1, n)$.
 If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.
z(*ldz*,*). The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
 If *jobz* = 'N', then *z* is not referenced.

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *spev* interface are the following:

ap Holds the array *A* of size $(n*(n+1)/2)$.
w Holds the vector with the number of elements *n*.
z Holds the matrix *Z* of size (n, n) .
uplo Must be 'U' or 'L'. The default value is 'U'.
jobz Restored based on the presence of the argument *z* as follows: *jobz* = 'V', if *z* is present, *jobz* = 'N', if *z* is omitted.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call hpev(ap, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>job</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A.</p> <p>If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A ($n \geq 0$).</p>
<i>ap</i>	<p>COMPLEX for chpev</p> <p>DOUBLE COMPLEX for zhpev.</p> <p>Array <i>ap</i>(*) contains the packed upper or lower triangle of Hermitian matrix A, as specified by <i>uplo</i>.</p> <p>The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p>
<i>work</i>	<p>COMPLEX for chpev</p> <p>DOUBLE COMPLEX for zhpev.</p> <p>(*) is a workspace array, size at least $\max(1, 2n-1)$.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>.</p> <p>Constraints:</p> <p>if <i>jobz</i> = 'N', then $ldz \geq 1$;</p> <p>if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.</p>
<i>rwork</i>	<p>REAL for chpev</p> <p>DOUBLE PRECISION for zhpev.</p> <p>Workspace array, size at least $\max(1, 3n-2)$.</p>

Output Parameters

<i>w</i>	<p>REAL for <code>chpev</code></p> <p>DOUBLE PRECISION for <code>zhpev</code>.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If $info = 0$, <i>w</i> contains the eigenvalues of the matrix <i>A</i> in ascending order.</p>
<i>z</i>	<p>COMPLEX for <code>chpev</code></p> <p>DOUBLE COMPLEX for <code>zhpev</code>.</p> <p>Array <i>z</i>(<i>ldz</i>,*).</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If $jobz = 'V'$, then if $info = 0$, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i>, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with $w(i)$.</p> <p>If $jobz = 'N'$, then <i>z</i> is not referenced.</p>
<i>ap</i>	<p>On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the <i>i</i>-th parameter had an illegal value.</p> <p>If $info = i$, then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpev` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

```
call sspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call dspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
call spevd(ap, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as:

$$A = Z \Lambda Z^T.$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	REAL for <i>sspevd</i> DOUBLE PRECISION for <i>dspevd</i> Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be $\max(1, n*(n+1)/2)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;
 if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $lwork \geq 2*n$;
 if $jobz = 'V'$ and $n > 1$, then
 $lwork \geq n^2 + 6*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w, z

REAL for *sspevd*

DOUBLE PRECISION for *dspevd*

Arrays:

$w(*)$, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

$z(ldz,*)$.

The second dimension of *z* must be: at least 1 if $jobz = 'N'$; at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. If $jobz = 'N'$, then *z* is not referenced.

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spevd` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [hpevd](#).

See also [syevd](#) for matrices held in full storage, and [sbevd](#) for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

```
call chpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call hpevd(ap, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z\Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	COMPLEX for chpevd DOUBLE COMPLEX for zhpevd Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array z .

Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;

if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork

REAL for *chpevd*

DOUBLE PRECISION for *zhpevd*

Workspace array, its dimension $\max(1, lrwork)$.

lrwork

INTEGER.

The dimension of the array *rwork*. Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>w</i>	<p>REAL for <code>chpevd</code></p> <p>DOUBLE PRECISION for <code>zhpevd</code></p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i>.</p>
<i>z</i>	<p>COMPLEX for <code>chpevd</code></p> <p>DOUBLE COMPLEX for <code>zhpevd</code></p> <p>Array, size (<i>ldz</i>, *).</p> <p>The second dimension of <i>z</i> must be:</p> <p>at least 1 if <i>jobz</i> = 'N';</p> <p>at least $\max(1, n)$ if <i>jobz</i> = 'V'.</p> <p>If <i>jobz</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i>.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>ap</i>	<p>On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i>.</p>
<i>work</i> (1)	<p>On exit, if <i>info</i> = 0, then <i>work</i>(1) returns the required minimal size of <i>lwork</i>.</p>
<i>rwork</i> (1)	<p>On exit, if <i>info</i> = 0, then <i>rwork</i>(1) returns the required minimal size of <i>lrwork</i>.</p>
<i>iwork</i> (1)	<p>On exit, if <i>info</i> = 0, then <i>iwork</i>(1) returns the required minimal size of <i>liwork</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = <i>i</i>, then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpevd` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .

<code>z</code>	Holds the matrix Z of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of `lwork` (`liwork` or `lrwork`) for the first run or set `lwork = -1` (`liwork = -1`, `lrwork = -1`).

If you choose the first option and set any of admissible `lwork` (`liwork` or `lrwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`) on exit. Use this value (`work(1)`, `iwork(1)`, `rwork(1)`) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [spevd](#).

See also [heevd](#) for matrices held in full storage, and [hbevd](#) for banded matrices.

?spevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
call sspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)
```

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)
```

```
call spevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i>.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>ap, work</i>	<p>REAL for <i>sspevx</i></p> <p>DOUBLE PRECISION for <i>dspevx</i></p> <p>Arrays:</p> <p>Array <i>ap</i>(*) contains the packed upper or lower triangle of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>work</i>(*) is a workspace array, size at least $\max(1, 8n)$.</p>
<i>vl, vu</i>	<p>REAL for <i>sspevx</i></p> <p>DOUBLE PRECISION for <i>dspevx</i></p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>REAL for <i>sspevx</i></p> <p>DOUBLE PRECISION for <i>dspevx</i></p> <p>The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.</p>

ldz INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then $ldz \geq 1$;

if *jobz* = 'V', then $ldz \geq \max(1, n)$.

iwork INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

ap On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V' the exact value of *m* is not known in advance..

w, z REAL for *sspevx*

DOUBLE PRECISION for *dspevx*

Arrays:

w(*), size at least $\max(1, n)$.

If *info* = 0, contains the selected eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least $\max(1, m)$.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail INTEGER.

Array, size at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array *ifail*.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spevx` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector with the number of elements n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -HUGE(vl)$.
<i>vu</i>	Default value for this element is $vu = HUGE(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \lambda_{amch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2*\epsilon_{lamch}('S')$.

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
call chpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
rwork, iwork, ifail, info)
```

```
call zhpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
rwork, iwork, ifail, info)
```

```
call hpevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	COMPLEX for chpevx DOUBLE COMPLEX for zhpevx Arrays: Array <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.

	<i>work</i> (*) is a workspace array, size at least $\max(1, 2n)$.
<i>vl, vu</i>	REAL for <i>chpevx</i> DOUBLE PRECISION for <i>zhpevx</i> If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for <i>chpevx</i> DOUBLE PRECISION for <i>zhpevx</i> The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for <i>chpevx</i> DOUBLE PRECISION for <i>zhpevx</i> Workspace array, size at least $\max(1, 7n)$.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, if <i>range</i> = 'I', $m = iu - il + 1$, and if <i>range</i> = 'V' the exact value of <i>m</i> is not known in advance..
<i>w</i>	REAL for <i>chpevx</i> DOUBLE PRECISION for <i>zhpevx</i> Array, size at least $\max(1, n)$.

If $info = 0$, contains the selected eigenvalues of the matrix A in ascending order.

z COMPLEX for `chpevx`
 DOUBLE COMPLEX for `zhpevx`

Array $z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$ifail$ INTEGER.

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpevx` interface are the following:

ap	Holds the array A of size $(n*(n+1)/2)$.
w	Holds the vector with the number of elements n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.
$ifail$	Holds the vector with the number of elements n .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{?lamch}('S')$.

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call sbev(ab, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	<p>INTEGER. The number of super- or sub-diagonals in <i>A</i></p> <p>($kd \geq 0$).</p>
<i>ab, work</i>	<p>REAL for <i>ssbev</i></p> <p>DOUBLE PRECISION for <i>dsbev</i>.</p> <p>Arrays:</p> <p><i>ab(lda,*)</i> is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>work</i> (*) is a workspace array.</p> <p>The dimension of <i>work</i> must be at least $\max(1, 3n-2)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd + 1$.
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>.</p> <p>Constraints:</p> <p>if <i>jobz</i> = 'N', then $ldz \geq 1$;</p> <p>if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.</p>

Output Parameters

<i>w, z</i>	<p>REAL for <i>ssbev</i></p> <p>DOUBLE PRECISION for <i>dsbev</i></p> <p>Arrays:</p> <p><i>w</i>(*), size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.</p> <p><i>z</i>(<i>ldz</i>,*).</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i>, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>).</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
-------------	---

<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form (see the description of ?sbtrd).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbevd` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?hbevd

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call hbevd(ab, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed.
-------------	---

	If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, work</i>	COMPLEX for chbev DOUBLE COMPLEX for zhbev. Arrays: <i>ab(lda,*)</i> is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd + 1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for chbev DOUBLE PRECISION for zhbev Workspace array, size at least $\max(1, 3n-2)$.

Output Parameters

<i>w</i>	REAL for chbev DOUBLE PRECISION for zhbev Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for chbev DOUBLE COMPLEX for zhbev. Array <i>z(ldz,*)</i> . The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>).

If `jobz = 'N'`, then `z` is not referenced.

`ab` On exit, this array is overwritten by the values generated during the reduction to tridiagonal form(see the description of [hbtrd](#)).

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*th parameter had an illegal value.

If `info = i`, then the algorithm failed to converge;

i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbev` interface are the following:

`ab` Holds the array *A* of size $(kd+1, n)$.

`w` Holds the vector with the number of elements *n*.

`z` Holds the matrix *Z* of size (n, n) .

`uplo` Must be 'U' or 'L'. The default value is 'U'.

`jobz` Restored based on the presence of the argument `z` as follows:

`jobz = 'V'`, if `z` is present,

`jobz = 'N'`, if `z` is omitted.

?sbevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

```
call ssbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dsbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call sbevd(ab, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix *A*. In other words, it can compute the spectral factorization of *A* as:

$$A = Z \Lambda Z^T$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	<p>INTEGER. The number of super- or sub-diagonals in <i>A</i></p> <p>($kd \geq 0$).</p>
<i>ab, work</i>	<p>REAL for <i>ssbevd</i></p> <p>DOUBLE PRECISION for <i>dsbevd</i>.</p> <p>Arrays:</p> <p><i>ab(lda,*)</i> is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd+1$.
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>.</p> <p>Constraints:</p> <p>if <i>jobz</i> = 'N', then $ldz \geq 1$;</p> <p>if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>Constraints:</p> <p>if $n \leq 1$, then $lwork \geq 1$;</p> <p>if <i>jobz</i> = 'N' and $n > 1$, then $lwork \geq 2n$;</p> <p>if <i>jobz</i> = 'V' and $n > 1$, then $lwork \geq 2*n^2 + 5*n + 1$.</p>

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*. Constraints: if $n \leq 1$, then $liwork < 1$; if $job = 'N'$ and $n > 1$, then $liwork < 1$; if $job = 'V'$ and $n > 1$, then $liwork < 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w, z REAL for `ssbevd`
DOUBLE PRECISION for `dsbevd`

Arrays:

$w(*)$, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

$z(ldz,*)$.

The second dimension of *z* must be:

at least 1 if $job = 'N'$;

at least $\max(1, n)$ if $job = 'V'$.

If $job = 'V'$, then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue $w(i)$.

If $job = 'N'$, then *z* is not referenced.

ab On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

work(1) On exit, if $lwork > 0$, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if $liwork > 0$, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbevd` interface are the following:

<code>ab</code>	Holds the array A of size $(kd+1, n)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>jobz</code>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If any of admissible `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if `work` (`liwork`) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [hbevd](#).

See also [syevd](#) for matrices held in full storage, and [spevd](#) for matrices held in packed storage.

?hbevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

call hbevd(*ab*, *w* [,*uplo*] [,*z*] [,*info*])

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix A . In other words, it can compute the spectral factorization of A as: $A = Z*\Lambda*Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A*z_i = \lambda_i*z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
<i>ab, work</i>	COMPLEX for <code>chbevd</code> DOUBLE COMPLEX for <code>zhbevd</code> . Arrays: <i>ab</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>lwork</i>	INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n^2$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork

REAL for chbevd

DOUBLE PRECISION for zhbevd

Workspace array, size at least *lrwork*.

lrwork

INTEGER.

The dimension of the array *rwork*.

Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER. Workspace array, size $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w

REAL for chbevd

DOUBLE PRECISION for zhbevd

Array, size at least $\max(1, n)$.

	If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>z</i>	COMPLEX for <i>chbevd</i> DOUBLE COMPLEX for <i>zhbevd</i> Array, size (<i>ldz</i> ,*). The second dimension of <i>z</i> must be: at least 1 if <i>jobz</i> = 'N'; at least $\max(1, n)$ if <i>jobz</i> = 'V'. If <i>jobz</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . The <i>i</i> -th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue $w(i)$. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then the real part of <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if <i>lrwork</i> > 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hbevd* interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size (<i>kd</i> +1, <i>n</i>).
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present,

$jobz = 'N'$, if z is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\epsilon)\|A\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ ($liwork$ or $lrwork$) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible $lwork$ ($liwork$ or $lrwork$) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$, $rwork$) on exit. Use this value ($work(1)$, $iwork(1)$, $rwork(1)$) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1$, $lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$, $rwork$). This operation is called a workspace query.

Note that if you set $lwork$ ($liwork$, $lrwork$) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [sbevd](#).

See also [heevd](#) for matrices held in full storage, and [hpevd](#) for matrices held in packed storage.

?sbevz

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
call ssbevz(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, iwork, ifail, info)
```

```
call dsbevz(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, iwork, ifail, info)
```

```
call sbevz(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

$jobz$ CHARACTER*1. Must be 'N' or 'V'.
 If $jobz = 'N'$, then only eigenvalues are computed.
 If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$.

If *range* = 'I', the routine computes eigenvalues with indices in range *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', *ab* stores the upper triangular part of *A*.

If *uplo* = 'L', *ab* stores the lower triangular part of *A*.

n INTEGER. The order of the matrix *A* ($n \geq 0$).

kd INTEGER. The number of super- or sub-diagonals in *A* ($kd \geq 0$).

ab, work REAL for *ssbev*x
DOUBLE PRECISION for *dsbev*x.

Arrays:
Arrays:
Array *ab(lda,*)* contains either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.
The second dimension of *ab* must be at least $\max(1, n)$.
work (*) is a workspace array.
The dimension of *work* must be at least $\max(1, 7n)$.

ldab INTEGER. The leading dimension of *ab*; must be at least $kd + 1$.

vl, vu REAL for *ssbev*x
DOUBLE PRECISION for *dsbev*x.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for *chpev*x
DOUBLE PRECISION for *zhpev*x

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldq, ldz

INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively.

Constraints:

$ldq \geq 1, ldz \geq 1;$

If *jobz* = 'V', then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$.

iwork

INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

q

REAL for *ssbevxdouble precision* for *dsbevxdouble precision*.

Array, size (*ldz*,*n*).

If *jobz* = 'V', the *n*-by-*n* orthogonal matrix is used in the reduction to tridiagonal form.

If *jobz* = 'N', the array *q* is not referenced.

m

INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V', the exact value of *m* is not known in advance.

w, z

REAL for *ssbevxdouble precision*

DOUBLE PRECISION for *dsbevxdouble precision*

Arrays:

w(*), size at least $\max(1, n)$. The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least $\max(1, m)$.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

<i>ifail</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices the eigenvectors that failed to converge.</p> <p>If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, then <i>i</i> eigenvectors failed to converge; their indices are stored in the array <i>ifail</i>.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbevz` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size $(kd+1, n)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector with the number of elements <i>n</i> .
<i>q</i>	Holds the matrix <i>Q</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	<p>Restored based on the presence of the argument <i>z</i> as follows:</p> <p><i>jobz</i> = 'V', if <i>z</i> is present,</p> <p><i>jobz</i> = 'N', if <i>z</i> is omitted</p> <p>Note that there will be an error condition if either <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.</p>
<i>range</i>	<p>Restored based on the presence of arguments <i>vl</i>, <i>vu</i>, <i>il</i>, <i>iu</i> as follows:</p> <p><i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present,</p> <p><i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present,</p>

$range = 'A'$, if none of vl , vu , il , iu is present,

Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \lambda_{\text{amch}}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \lambda_{\text{amch}}('S')$.

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
call chbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call hbevx(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
 If $job = 'N'$, then only eigenvalues are computed.
 If $job = 'V'$, then eigenvalues and eigenvectors are computed.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.
 If $range = 'A'$, the routine computes all eigenvalues.
 If $range = 'V'$, the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$.
 If $range = 'I'$, the routine computes eigenvalues with indices il to iu .

uplo CHARACTER*1. Must be 'U' or 'L'.

	If $uplo = 'U'$, ab stores the upper triangular part of A .
	If $uplo = 'L'$, ab stores the lower triangular part of A .
n	INTEGER. The order of the matrix A ($n \geq 0$).
kd	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
$ab, work$	COMPLEX for <code>chbev</code> DOUBLE COMPLEX for <code>zhbev</code> .
	Arrays:
	$ab(lda,*)$ is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by $uplo$) in band storage format.
	The second dimension of ab must be at least $\max(1, n)$.
	$work(*)$ is a workspace array.
	The dimension of $work$ must be at least $\max(1, n)$.
$ldab$	INTEGER. The leading dimension of ab ; must be at least $kd + 1$.
vl, vu	REAL for <code>chbev</code> DOUBLE PRECISION for <code>zhbev</code> .
	If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.
	Constraint: $vl < vu$.
	If $range = 'A'$ or $'I'$, vl and vu are not referenced.
il, iu	INTEGER.
	If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.
	Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
	If $range = 'A'$ or $'V'$, il and iu are not referenced.
$abstol$	REAL for <code>chbev</code> DOUBLE PRECISION for <code>zhbev</code> .
	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
ldq, ldz	INTEGER. The leading dimensions of the output arrays q and z , respectively.
	Constraints:
	$ldq \geq 1$, $ldz \geq 1$;
	If $jobz = 'V'$, then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$.
$rwork$	REAL for <code>chbev</code> DOUBLE PRECISION for <code>zhbev</code>

Workspace array, size at least $\max(1, 7n)$.

iwork INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

q COMPLEX for *chbevxd* DOUBLE COMPLEX for *zhbevxd*.

Array, size (ldz, n) .

If *jobz* = 'V', the *n*-by-*n* unitary matrix is used in the reduction to tridiagonal form.

If *jobz* = 'N', the array *q* is not referenced.

m INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$.

If *range* = 'A', *m* = *n*, if *range* = 'I', *m* = *iu-il*+1, and if *range* = 'V', the exact value of *m* is not known in advance..

w REAL for *chbevxd*

DOUBLE PRECISION for *zhbevxd*

Array, size at least $\max(1, n)$. The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z COMPLEX for *chbevxd*

DOUBLE COMPLEX for *zhbevxd*.

Array *z*(*ldz*, *).

The second dimension of *z* must be at least $\max(1, m)$.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail INTEGER.

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbevx` interface are the following:

<i>ab</i>	Holds the array A of size $(kd+1, n)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector with the number of elements n .
<i>q</i>	Holds the matrix Q of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted Note that there will be an error condition if either $ifail$ or q is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present,

Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{?lamch}('S')$.

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstev(jobz, n, d, e, z, ldz, work, info)
call dstev(jobz, n, d, e, z, ldz, work, info)
call stev(d, e [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A .

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>d, e, work</i>	REAL for sstev DOUBLE PRECISION for dstev. Arrays: Array <i>d</i> (*) contains the n diagonal elements of the tridiagonal matrix A . The size of <i>d</i> must be at least $\max(1, n)$. Array <i>e</i> (*) contains the $n-1$ subdiagonal elements of the tridiagonal matrix A . The size of <i>e</i> must be at least $\max(1, n)$. The n -th element of this array is used as workspace. <i>work</i> (*) is a workspace array.

The dimension of *work* must be at least $\max(1, 2n-2)$.

If *jobz* = 'N', *work* is not referenced.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V' then $ldz \geq \max(1, n)$.

Output Parameters

d On exit, if *info* = 0, contains the eigenvalues of the matrix *A* in ascending order.

z REAL for *sstev*
DOUBLE PRECISION for *dstev*
Array, size (*ldz*,*).
The second dimension of *z* must be at least $\max(1, n)$.
If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*).
If *jobz* = 'N', then *z* is not referenced.

e On exit, this array is overwritten with intermediate results.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.
If *info* = *i*, then the algorithm failed to converge;
i elements of *e* did not converge to zero.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *stev* interface are the following:

d Holds the vector of length *n*.
e Holds the vector of length *n*.
z Holds the matrix *Z* of size (*n*, *n*).
jobz Restored based on the presence of the argument *z* as follows:
jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

?stevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

```
call sstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call stevd(d, e [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization of T as: $T = Z^* \Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$T^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

There is no complex analogue of this routine.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for <i>sstevd</i> DOUBLE PRECISION for <i>dstevd</i> . Arrays: <i>d</i> (*) contains the n diagonal elements of the tridiagonal matrix T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the $n-1$ off-diagonal elements of T . The dimension of <i>e</i> must be at least $\max(1, n)$. The n -th element of this array is used as workspace. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least <i>lwork</i> .
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: <i>ldz</i> ≥ 1 if <i>job</i> = 'N'; <i>ldz</i> $\geq \max(1, n)$ if <i>job</i> = 'V'.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> .

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $lwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $lwork \geq n^2 + 4*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

d On exit, if $info = 0$, contains the eigenvalues of the matrix *T* in ascending order.

See also *info*.

z REAL for *sstevd*
DOUBLE PRECISION for *dstevd*

Array, size (*ldz*,*) .

The second dimension of *z* must be:

at least 1 if $jobz = 'N'$;

at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *T*.

If $jobz = 'N'$, then *z* is not referenced.

e On exit, this array is overwritten with intermediate results.

work(1) On exit, if $lwork > 0$, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if $liwork > 0$, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stevd` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) .
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_j is an exact eigenvalue, and μ_j is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run, or set $lwork = -1$ ($liwork = -1$).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call stevx(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval: $vl < w(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>d, e, work</i>	REAL for sstevx DOUBLE PRECISION for dstevx. Arrays: <i>d</i> (*) contains the n diagonal elements of the tridiagonal matrix A . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the $n-1$ subdiagonal elements of A . The dimension of <i>e</i> must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 5n)$.
<i>vl, vu</i>	REAL for sstevx DOUBLE PRECISION for dstevx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or $'I'$, vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If $range = 'A'$ or $'V'$, il and iu are not referenced.

$abstol$

REAL for `sstevx`

DOUBLE PRECISION for `dstevx`. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimensions of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, then $ldz \geq \max(1, n)$.

$iwork$

INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$.

If $range = 'A'$, $m = n$, if $range = 'I'$, $m = iu - il + 1$, and if $range = 'V'$ the exact value of m is unknown.

w, z

REAL for `sstevx`

DOUBLE PRECISION for `dstevx`.

Arrays:

$w(*)$, size at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues of the matrix A in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

d, e	On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.
$ifail$	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.</p> <p>If $jobz = 'N'$, then $ifail$ is not referenced.</p>
$info$	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stevx` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
w	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.
$ifail$	Holds the vector of length n .
vl	Default value for this element is $vl = -\text{HUGE}(vl)$.
vu	Default value for this element is $vu = \text{HUGE}(vu)$.
il	Default value for this argument is $il = 1$.
iu	Default value for this argument is $iu = n$.
$abstol$	Default value for this element is $abstol = 0.0_WP$.
$jobz$	<p>Restored based on the presence of the argument z as follows:</p> <p>$jobz = 'V'$, if z is present,</p> <p>$jobz = 'N'$, if z is omitted</p> <p>Note that there will be an error condition if $ifail$ is present and z is omitted.</p>

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,
range = 'I', if one of or both *il* and *iu* are present,
range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot |A|_1$ is used instead. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to $2 * \text{lamch}('S')$.

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

```
call sstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call dstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

```
call stevr(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls [stemr](#) to compute the eigenspectrum using Relatively Robust Representations. [stegr](#) computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L * D * L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the *i*-th unreduced block of *T*:

- Compute $T - \sigma_i = L_i * D_i * L_i^T$, such that $L_i * D_i * L_i^T$ is a relatively robust representation.
- Compute the eigenvalues, λ_j , of $L_i * D_i * L_i^T$ to high relative accuracy by the *dqds* algorithm.
- If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to Step (a).
- Given the approximate eigenvalue λ_j of $L_i * D_i * L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?stevr` calls `stemr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls `stebz` and `stein` on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then only eigenvalues are computed.</p> <p>If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w(i)$ in the half-open interval:</p> $vl < w(i) \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p> <p>For <i>range</i> = 'V' or 'I' and $iu - il < n - 1$, <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.</p>
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	<p>REAL for <code>sstevr</code></p> <p>DOUBLE PRECISION for <code>dstevr</code>.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the <i>n</i> diagonal elements of the tridiagonal matrix T.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the <i>n</i>-1 subdiagonal elements of A.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n-1)$. The <i>n</i>-th element of this array is used as workspace.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>vl, vu</i>	<p>REAL for <code>sstevr</code></p> <p>DOUBLE PRECISION for <code>dstevr</code>.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	REAL for <code>sstevr</code>

DOUBLE PRECISION for `dstevr`.

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If `jobz = 'V'`, the eigenvalues and eigenvectors output have residual norms bounded by `abstol`, and the dot products between different eigenvectors are bounded by `abstol`. If `abstol < n * eps * ||T||`, then `n * eps * ||T||` will be used in its place, where `eps` is the machine precision, and `||T||` is the 1-norm of the matrix `T`. The eigenvalues are computed to an accuracy of `eps * ||T||` irrespective of `abstol`.

If high relative accuracy is important, set `abstol` to `?lamch('S')`.

`ldz`

INTEGER. The leading dimension of the output array `z`.

Constraints:

`ldz ≥ 1` if `jobz = 'N'`;

`ldz ≥ max(1, n)` if `jobz = 'V'`.

`lwork`

INTEGER.

The dimension of the array `work`. Constraint:

`lwork ≥ max(1, 20*n)`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the required sizes of the `work` and `iwork` arrays, returns these values as the first entries of the `work` and `iwork` arrays, and no error message related to `lwork` or `liwork` is issued by [xerbla](#). See *Application Notes* for details.

`iwork`

INTEGER.

Workspace array, its dimension `max(1, liwork)`.

`liwork`

INTEGER.

The dimension of the array `iwork`,

`liwork ≥ max(1, 10*n)`.

If `liwork = -1`, then a workspace query is assumed; the routine only calculates the required sizes of the `work` and `iwork` arrays, returns these values as the first entries of the `work` and `iwork` arrays, and no error message related to `lwork` or `liwork` is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

`m`

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If `range = 'A'`, `m = n`, if `range = 'I'`, `m = iu-il+1`, and if `range = 'V'` the exact value of `m` is unknown..

`w, z`

REAL for `sstevr`

DOUBLE PRECISION for `dstevr`.

Arrays:

$w(*)$, size at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues of the matrix T in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

d, e

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

$isuppz$

INTEGER.

Array, size at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz(2i-1)$ through $isuppz(2i)$.

Implemented only for $range = 'A'$ or $'I'$ and $iu-il = n-1$.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$iwork(1)$

On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, an internal error has occurred.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `stevr` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
w	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.

<i>isuppz</i>	Holds the vector of length $(2*n)$, where the values $(2*m)$ are significant.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

Normal execution of the routine `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table "Driver Routines for Solving Nonsymmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
gees	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
geesx	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
geev	Computes the eigenvalues and left and right eigenvectors of a general matrix.
geevx	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

```
call sgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
```

```
call dgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork, info)
```

```
call cgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
```

```
call zgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork, bwork, info)
```

```
call gees(a, wr, wi [,vs] [,select] [,sdim] [,info])
```

```
call gees(a, w [,vs] [,select] [,sdim] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{bmatrix} \dots & & \\ & \dots & \\ & & \dots \end{bmatrix}$$

where $b^*c < 0$. The eigenvalues of such a block are $a \pm i\sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

<i>jobvs</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvs</i> = 'N', then Schur vectors are not computed.</p> <p>If <i>jobvs</i> = 'V', then Schur vectors are computed.</p>
<i>sort</i>	<p>CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.</p> <p>If <i>sort</i> = 'N', then eigenvalues are not ordered.</p> <p>If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).</p>
<i>select</i>	<p>LOGICAL FUNCTION of two REAL arguments for real flavors.</p> <p>LOGICAL FUNCTION of one COMPLEX argument for complex flavors.</p> <p><i>select</i> must be declared EXTERNAL in the calling subroutine.</p> <p>If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form.</p> <p>If <i>sort</i> = 'N', <i>select</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>An eigenvalue $wr(j) + \sqrt{-1} * wi(j)$ is selected if <i>select</i>(<i>wr</i>(<i>j</i>), <i>wi</i>(<i>j</i>)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p><i>For complex flavors:</i></p> <p>An eigenvalue <i>w</i>(<i>j</i>) is selected if <i>select</i>(<i>w</i>(<i>j</i>)) is true.</p> <p>Note that a selected complex eigenvalue may no longer satisfy <i>select</i>(<i>wr</i>(<i>j</i>), <i>wi</i>(<i>j</i>)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> may be set to <i>n</i>+2 (see <i>info</i> below).</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>a, work</i>	<p>REAL for sgees</p> <p>DOUBLE PRECISION for dgees</p> <p>COMPLEX for cgees</p> <p>DOUBLE COMPLEX for zgees.</p> <p>Arrays:</p> <p><i>a</i>(<i>lda</i>,*) is an array containing the <i>n</i>-by-<i>n</i> matrix <i>A</i>.</p> <p>The second dimension of <i>a</i> must be at least max(1, <i>n</i>).</p> <p><i>work</i> is a workspace array, its dimension max(1, <i>lwork</i>).</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. Must be at least max(1, <i>n</i>).</p>
<i>ldvs</i>	<p>INTEGER. The leading dimension of the output array <i>vs</i>. Constraints:</p> <p><i>ldvs</i> ≥ 1;</p>

$ldvs \geq \max(1, n)$ if $jobvs = 'V'$.

lwork

INTEGER.

The dimension of the array *work*.

Constraint:

$lwork \geq \max(1, 3n)$ for real flavors;

$lwork \geq \max(1, 2n)$ for complex flavors.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork

REAL for *cgees*

DOUBLE PRECISION for *zgees*

Workspace array, size at least $\max(1, n)$. Used in complex flavors only.

bwork

LOGICAL. Workspace array, size at least $\max(1, n)$. Not referenced if $sort = 'N'$.

Output Parameters

a

On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim

INTEGER.

If $sort = 'N'$, $sdim = 0$.

If $sort = 'S'$, $sdim$ is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi

REAL for *sgees*

DOUBLE PRECISION for *dgees*

Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for *cgees*

DOUBLE COMPLEX for *zgees*.

Array, size at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs

REAL for *sgees*

DOUBLE PRECISION for *dgees*

COMPLEX for *cgees*

DOUBLE COMPLEX for *zgees*.

Array *vs(ldvs,*)*; the second dimension of *vs* must be at least $\max(1, n)$.

If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.

If *jobvs* = 'N', *vs* is not referenced.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

i ≤ *n*:

the *QR* algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* = 'V', *vs* contains the matrix which reduces *A* to its partially converged Schur form;

i = *n*+1:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

i = *n*+2:

after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *gees* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>wr</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>wi</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>w</i>	Holds the vector of length <i>n</i> . Used in complex flavors only.
<i>vs</i>	Holds the matrix <i>VS</i> of size (<i>n</i> , <i>n</i>).
<i>jobvs</i>	Restored based on the presence of the argument <i>vs</i> as follows: <i>jobvs</i> = 'V', if <i>vs</i> is present, <i>jobvs</i> = 'N', if <i>vs</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows:

`sort = 'S'`, if *select* is present,
`sort = 'N'`, if *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde,
rcondv, work, lwork, iwork, liwork, bwork, info)
```

```
call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde,
rcondv, work, lwork, iwork, liwork, bwork, info)
```

```
call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv,
work, lwork, rwork, bwork, info)
```

```
call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv,
work, lwork, rwork, bwork, info)
```

```
call geesx(a, wr, wi [,vs] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

```
call geesx(a, w [,vs] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

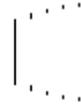
Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real-Schur/Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [LUG], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form



where $b*c < 0$. The eigenvalues of such a block are $a \pm i\sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

<i>jobvs</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvs</i> = 'N', then Schur vectors are not computed.</p> <p>If <i>jobvs</i> = 'V', then Schur vectors are computed.</p>
<i>sort</i>	<p>CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.</p> <p>If <i>sort</i> = 'N', then eigenvalues are not ordered.</p> <p>If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).</p>
<i>select</i>	<p>LOGICAL FUNCTION of two REAL arguments for real flavors.</p> <p>LOGICAL FUNCTION of one COMPLEX argument for complex flavors.</p> <p><i>select</i> must be declared EXTERNAL in the calling subroutine.</p> <p>If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form.</p> <p>If <i>sort</i> = 'N', <i>select</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>An eigenvalue $wr(j)+sqrt(-1)*wi(j)$ is selected if <i>select</i>(<i>wr</i>(<i>j</i>), <i>wi</i>(<i>j</i>)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p><i>For complex flavors:</i></p> <p>An eigenvalue <i>w</i>(<i>j</i>) is selected if <i>select</i>(<i>w</i>(<i>j</i>)) is true.</p> <p>Note that a selected complex eigenvalue may no longer satisfy <i>select</i>(<i>wr</i>(<i>j</i>), <i>wi</i>(<i>j</i>))= .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> may be set to <i>n</i>+2 (see <i>info</i> below).</p>
<i>sense</i>	<p>CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = 'N', none are computed;</p> <p>If <i>sense</i> = 'E', computed for average of selected eigenvalues only;</p> <p>If <i>sense</i> = 'V', computed for selected right invariant subspace only;</p> <p>If <i>sense</i> = 'B', computed for both.</p> <p>If <i>sense</i> is 'E', 'V', or 'B', then <i>sort</i> must equal 'S'.</p>

n INTEGER. The order of the matrix *A* ($n \geq 0$).

a, work REAL for sgeesx
 DOUBLE PRECISION for dgeesx
 COMPLEX for cgeesx
 DOUBLE COMPLEX for zgeesx.

Arrays:
a(lda,)* is an array containing the *n*-by-*n* matrix *A*.
 The second dimension of *a* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, n)$.

ldvs INTEGER. The leading dimension of the output array *vs*. Constraints:
 $ldvs \geq 1$;
 $ldvs \geq \max(1, n)$ if *jobvs* = 'V'.

lwork INTEGER.
 The dimension of the array *work*. Constraint:
 $lwork \geq \max(1, 3n)$ for real flavors;
 $lwork \geq \max(1, 2n)$ for complex flavors.
 Also, if *sense* = 'E', 'V', or 'B', then
 $lwork \geq n+2*sdim*(n-sdim)$ for real flavors;
 $lwork \geq 2*sdim*(n-sdim)$ for complex flavors;
 where *sdim* is the number of selected eigenvalues computed by this routine.
 Note that $2*sdim*(n-sdim) \leq n*n/2$. Note also that an error is only returned if $lwork < \max(1, 2*n)$, but if *sense* = 'E', or 'V', or 'B' this may not be large enough.
 For good performance, *lwork* must generally be larger.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates upper bound on the optimal size of the array *work*, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

iwork INTEGER.
 Workspace array, size (*liwork*). Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

liwork INTEGER.
 The dimension of the array *iwork*. Used in real flavors only.
 Constraint:
 $liwork \geq 1$;

if *sense* = 'V' or 'B', $liwork \geq sdim * (n - sdim)$.

rwork

REAL for *cgeesx*

DOUBLE PRECISION for *zgeesx*

Workspace array, size at least $\max(1, n)$. Used in complex flavors only.

bwork

LOGICAL. Workspace array, size at least $\max(1, n)$. Not referenced if *sort* = 'N'.

Output Parameters

a

On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim

INTEGER.

If *sort* = 'N', *sdim* = 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi

REAL for *sgeesx*

DOUBLE PRECISION for *dgeesx*

Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for *cgeesx*

DOUBLE COMPLEX for *zgeesx*.

Array, size at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs

REAL for *sgeesx*

DOUBLE PRECISION for *dgeesx*

COMPLEX for *cgeesx*

DOUBLE COMPLEX for *zgeesx*.

Array *vs(ldvs,*)*; the second dimension of *vs* must be at least $\max(1, n)$.

If *jobvs* = 'V', *vs* contains the orthogonal/unitary matrix *Z* of Schur vectors.

If *jobvs* = 'N', *vs* is not referenced.

rconde, rcondv

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

If *sense* = 'E' or 'B', *rconde* contains the reciprocal condition number for the average of the selected eigenvalues.

	If <i>sense</i> = 'N' or 'V', <i>rconde</i> is not referenced.
	If <i>sense</i> = 'V' or 'B', <i>rcondv</i> contains the reciprocal condition number for the selected right invariant subspace.
	If <i>sense</i> = 'N' or 'E', <i>rcondv</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER.
	If <i>info</i> = 0, the execution is successful.
	If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.
	If <i>info</i> = <i>i</i> , and
	<i>i</i> ≤ <i>n</i> :
	the QR algorithm failed to compute all the eigenvalues; elements 1: <i>ilo</i> -1 and <i>i</i> +1: <i>n</i> of <i>wr</i> and <i>wi</i> (for real flavors) or <i>w</i> (for complex flavors) contain those eigenvalues which have converged; if <i>jobvs</i> = 'V', <i>vs</i> contains the transformation which reduces <i>A</i> to its partially converged Schur form;
	<i>i</i> = <i>n</i> +1:
	the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);
	<i>i</i> = <i>n</i> +2:
	after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy <i>select</i> = .TRUE.. This could also be caused by underflow due to scaling.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *geesx* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>wr</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>wi</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>w</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>vs</i>	Holds the matrix <i>VS</i> of size (<i>n</i> , <i>n</i>).
<i>jobvs</i>	Restored based on the presence of the argument <i>vs</i> as follows: <i>jobvs</i> = 'V', if <i>vs</i> is present, <i>jobvs</i> = 'N', if <i>vs</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present,

`sort = 'N'`, if `select` is omitted.

`sense` Restored based on the presence of arguments `rconde` and `rcondv` as follows:

`sense = 'B'`, if both `rconde` and `rcondv` are present,
`sense = 'E'`, if `rconde` is present and `rcondv` omitted,
`sense = 'V'`, if `rconde` is omitted and `rcondv` present,
`sense = 'N'`, if both `rconde` and `rcondv` are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If you choose the first option and set any of admissible `lwork` (or `liwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

```
call sgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call cgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call zgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call geev(a, wr, wi [,vl] [,vr] [,info])
call geev(a, w [,vl] [,vr] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector v of A satisfies

$$A * v = \lambda * v$$

where λ is its eigenvalue.

The left eigenvector u of A satisfies

$$u^H * A = \lambda * u^H$$

where u^H denotes the conjugate transpose of u . The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

<i>jobvl</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', then left eigenvectors of <i>A</i> are not computed.</p> <p>If <i>jobvl</i> = 'V', then left eigenvectors of <i>A</i> are computed.</p>
<i>jobvr</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', then right eigenvectors of <i>A</i> are not computed.</p> <p>If <i>jobvr</i> = 'V', then right eigenvectors of <i>A</i> are computed.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>a, work</i>	<p>REAL for <i>sggev</i></p> <p>DOUBLE PRECISION for <i>dgeev</i></p> <p>COMPLEX for <i>cgeev</i></p> <p>DOUBLE COMPLEX for <i>zgeev</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> is an array containing the <i>n</i>-by-<i>n</i> matrix <i>A</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. Must be at least $\max(1, n)$.</p>
<i>ldvl, ldvr</i>	<p>INTEGER. The leading dimensions of the output arrays <i>vl</i> and <i>vr</i>, respectively.</p> <p>Constraints:</p> <p>$ldvl \geq 1; ldvr \geq 1$.</p> <p>If <i>jobvl</i> = 'V', $ldvl \geq \max(1, n)$;</p> <p>If <i>jobvr</i> = 'V', $ldvr \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>Constraint for real flavors: $lwork \geq \max(1, 3n)$. If computing eigenvectors (<i>jobvl</i> = 'V' or <i>jobvr</i> = 'V'), $lwork \geq \max(1, 4n)$.</p> <p>Constraint for complex flavors: $lwork \geq \max(1, 2n)$.</p> <p>For good performance, <i>lwork</i> must generally be larger.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i>.</p>

work REAL for cgeev
 DOUBLE PRECISION for zgeev
 Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

a On exit, this array is overwritten.

wr, wi REAL for sgeev
 DOUBLE PRECISION for dgeev
 Arrays, size at least $\max(1, n)$ each.
 Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w COMPLEX for cgeev
 DOUBLE COMPLEX for zgeev.
 Array, size at least $\max(1, n)$.
 Contains the computed eigenvalues.

v1, vr REAL for sgeev
 DOUBLE PRECISION for dgeev
 COMPLEX for cgeev
 DOUBLE COMPLEX for zgeev.
Arrays:
 $v1(ldv1,*)$; the second dimension of *v1* must be at least $\max(1, n)$.
 If *jobv1* = 'N', *v1* is not referenced.
For real flavors:
 If the *j*-th eigenvalue is real, then $u_j = v1(:, j)$, the *j*-th column of *v1*.
 If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \sqrt{-1}$, $u_j = v1(:, j) + i*v1(:, j+1)$ and $u_{j+1} = v1(:, j) - i*v1(:, j+1)$.
For complex flavors:
 $u_j = v1(:, j)$, the *j*-th column of *v1*.
 $vr(ldvr,*)$; the second dimension of *vr* must be at least $\max(1, n)$.
 If *jobvr* = 'N', *vr* is not referenced.
For real flavors:
 If the *j*-th eigenvalue is real, then $v_j = vr(:, j)$, the *j*-th column of *vr*.
 If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \sqrt{-1}$, $v_j = vr(:, j) + i*vr(:, j+1)$ and $v_{j+1} = vr(:, j) - i*vr(:, j+1)$.
For complex flavors:

$v_j = vr(:, j)$, the j -th column of vr .

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*th parameter had an illegal value.
 If *info* = *i*, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements $i+1:n$ of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *geev* interface are the following:

a Holds the matrix A of size (n, n) .

wr Holds the vector of length n . Used in real flavors only.

wi Holds the vector of length n . Used in real flavors only.

w Holds the vector of length n . Used in complex flavors only.

vl Holds the matrix VL of size (n, n) .

vr Holds the matrix VR of size (n, n) .

jobvl Restored based on the presence of the argument *vl* as follows:
 $jobvl = 'V'$, if *vl* is present,
 $jobvl = 'N'$, if *vl* is omitted.

jobvr Restored based on the presence of the argument *vr* as follows:
 $jobvr = 'V'$, if *vr* is present,
 $jobvr = 'N'$, if *vr* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine exits immediately with an error and does not provide any information on the recommended workspace.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

```
call sgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo,
ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
```

```
call dgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo,
ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
```

```
call cgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi,
scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

```
call zgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi,
scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

```
call geevx(a, wr, wi [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [, rconde]
[,rcondv] [,info])
```

```
call geevx(a, w [,vl] [,vr] [,balanc] [,ilo] [,ihi] [,scale] [,abnrm] [,rconde] [,
rcondv] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

The right eigenvector v of A satisfies

$$A \cdot v = \lambda \cdot v$$

where λ is its eigenvalue.

The left eigenvector u of A satisfies

$$u^H A = \lambda u^H$$

where u^H denotes the conjugate transpose of u . The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D \cdot A \cdot \text{inv}(D)$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

<i>balanc</i>	<p>CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.</p> <p>If <i>balanc</i> = 'N', do not diagonally scale or permute;</p> <p>If <i>balanc</i> = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;</p> <p>If <i>balanc</i> = 'S', diagonally scale the matrix, i.e. replace <i>A</i> by $D*A*inv(D)$, where <i>D</i> is a diagonal matrix chosen to make the rows and columns of <i>A</i> more equal in norm. Do not permute;</p> <p>If <i>balanc</i> = 'B', both diagonally scale and permute <i>A</i>.</p> <p>Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.</p>
<i>jobvl</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', left eigenvectors of <i>A</i> are not computed;</p> <p>If <i>jobvl</i> = 'V', left eigenvectors of <i>A</i> are computed.</p> <p>If <i>sense</i> = 'E' or 'B', then <i>jobvl</i> must be 'V'.</p>
<i>jobvr</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', right eigenvectors of <i>A</i> are not computed;</p> <p>If <i>jobvr</i> = 'V', right eigenvectors of <i>A</i> are computed.</p> <p>If <i>sense</i> = 'E' or 'B', then <i>jobvr</i> must be 'V'.</p>
<i>sense</i>	<p>CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = 'N', none are computed;</p> <p>If <i>sense</i> = 'E', computed for eigenvalues only;</p> <p>If <i>sense</i> = 'V', computed for right eigenvectors only;</p> <p>If <i>sense</i> = 'B', computed for eigenvalues and right eigenvectors.</p> <p>If <i>sense</i> is 'E' or 'B', both left and right eigenvectors must also be computed (<i>jobvl</i> = 'V' and <i>jobvr</i> = 'V').</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).</p>
<i>a, work</i>	<p>REAL for <i>sggevx</i></p> <p>DOUBLE PRECISION for <i>dgeevx</i></p> <p>COMPLEX for <i>cgeevx</i></p> <p>DOUBLE COMPLEX for <i>zgeevx</i>.</p> <p>Arrays:</p> <p><i>a(lda,*)</i> is an array containing the <i>n</i>-by-<i>n</i> matrix <i>A</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, n)$.

ldvl, ldvr INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.

Constraints:

$ldvl \geq 1; ldvr \geq 1$.

If *jobvl* = 'V', $ldvl \geq \max(1, n)$;

If *jobvr* = 'V', $ldvr \geq \max(1, n)$.

lwork INTEGER.

The dimension of the array *work*.

For real flavors:

If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$, and if *jobvl* = 'V' or *jobvr* = 'V', $lwork \geq 3n$;

If *sense* = 'V' or 'B', $lwork \geq n*(n+6)$.

For good performance, *lwork* must generally be larger.

For complex flavors:

If *sense* = 'N' or 'E', $lwork \geq \max(1, 2n)$;

If *sense* = 'V' or 'B', $lwork \geq n^2+2n$. For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork REAL for *cgeevx*

DOUBLE PRECISION for *zgeevx*

Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

iwork INTEGER.

Workspace array, size at least $\max(1, 2n-2)$. Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a On exit, this array is overwritten.

If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, wi REAL for *sggevx*

DOUBLE PRECISION for *dgeevx*

Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for *cggeevx*

DOUBLE COMPLEX for *zgeevx*.

Array, size at least $\max(1, n)$. Contains the computed eigenvalues.

v1, vr

REAL for *sggeevx*

DOUBLE PRECISION for *dgeevx*

COMPLEX for *cggeevx*

DOUBLE COMPLEX for *zgeevx*.

Arrays:

v1(*ldv1*,*); the second dimension of *v1* must be at least $\max(1, n)$.

If *jobv1* = 'N', *v1* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then $u_j = v1(:, j)$, the *j*-th column of *v1*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, $u_j = v1(:, j) + i*v1(:, j+1)$ and $u_{j+1} = v1(:, j) - i*v1(:, j+1)$.

For complex flavors:

$u_j = v1(:, j)$, the *j*-th column of *v1*.

vr(*ldvr*,*); the second dimension of *vr* must be at least $\max(1, n)$.

If *jobvr* = 'N', *vr* is not referenced.

For real flavors:

If the *j*-th eigenvalue is real, then $v_j = vr(:, j)$, the *j*-th column of *vr*.

If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, $v_j = vr(:, j) + i*vr(:, j+1)$ and $v_{j+1} = vr(:, j) - i*vr(:, j+1)$.

For complex flavors:

$v_j = vr(:, j)$, the *j*-th column of *vr*.

ilo, ihi

INTEGER. *ilo* and *ihi* are integer values determined when *A* was balanced.

The balanced $A(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If *balanc* = 'N' or 'S', *ilo* = 1 and *ihi* = *n*.

scale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, size at least $\max(1, n)$. Details of the permutations and scaling factors applied when balancing *A*.

If $P(j)$ is the index of the row and column interchanged with row and column j , and $D(j)$ is the scaling factor applied to row and column j , then

$$\begin{aligned} \text{scale}(j) &= P(j), \text{ for } j = 1, \dots, ilo-1 \\ &= D(j), \text{ for } j = ilo, \dots, ihi \\ &= P(j) \text{ for } j = ihi+1, \dots, n. \end{aligned}$$

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

abnrm

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde, rcondv

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(1, n)$ each.

rconde(j) is the reciprocal condition number of the j -th eigenvalue.

rcondv(j) is the reciprocal condition number of the j -th right eigenvector.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th parameter had an illegal value.

If $info = i$, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1: $ilo-1$ and $i+1:n$ of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *geevx* interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>wr</i>	Holds the vector of length n . Used in real flavors only.
<i>wi</i>	Holds the vector of length n . Used in real flavors only.
<i>w</i>	Holds the vector of length n . Used in complex flavors only.
<i>vl</i>	Holds the matrix VL of size (n, n) .
<i>vr</i>	Holds the matrix VR of size (n, n) .
<i>scale</i>	Holds the vector of length n .

<i>rconde</i>	Holds the vector of length n .
<i>rcondv</i>	Holds the vector of length n .
<i>balanc</i>	Must be 'N', 'B', 'P' or 'S'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition - LAPACK Driver Routines

Table "Driver Routines for Singular Value Decomposition" lists the LAPACK driver routines that perform singular value decomposition for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?gejsv	Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.
?gesvj	Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

Routine Name	Operation performed
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Singular Value Decomposition - LAPACK Computational Routines

?gesvd

Computes the singular value decomposition of a general rectangular matrix.

Syntax

```
call sgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call dgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call cgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call zgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call gesvd(a, s [,u] [,vt] [,ww] [,job] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written as

$$A = U \Sigma V^T \text{ for real routines}$$

$$A = U \Sigma V^H \text{ for complex routines}$$

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns V^T (for real flavors) or V^H (for complex flavors), not V .

Input Parameters

jobu CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U .

If *jobu* = 'A', all m columns of U are returned in the array u ;

if *jobu* = 'S', the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array u ;

if *jobu* = 'O', the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array a ;

if *jobu* = 'N', no columns of U (no left singular vectors) are computed.

jobvt CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix V^T/V^H .

If *jobvt* = 'A', all n rows of V^T/V^H are returned in the array vt ;

if $jobvt = 'S'$, the first $\min(m,n)$ rows of V^T/V^H (the right singular vectors) are returned in the array vt ;
 if $jobvt = 'O'$, the first $\min(m,n)$ rows of V^T/V^H (the right singular vectors) are overwritten on the array a ;
 if $jobvt = 'N'$, no rows of V^T/V^H (no right singular vectors) are computed.
 $jobvt$ and $jobu$ cannot both be $'O'$.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for `sgesvd`
 DOUBLE PRECISION for `dgesvd`
 COMPLEX for `cgesvd`
 DOUBLE COMPLEX for `zgesvd`.

Arrays:

$a(lda,*)$ is an array containing the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array a .
 Must be at least $\max(1, m)$.

$ldu, ldvt$ INTEGER. The leading dimensions of the output arrays u and vt , respectively.

Constraints:

$ldu \geq 1$; $ldvt \geq 1$.

If $jobu = 'A'$ or $'S'$, $ldu \geq m$;

If $jobvt = 'A'$, $ldvt \geq n$;

If $jobvt = 'S'$, $ldvt \geq \min(m, n)$.

$lwork$ INTEGER.

The dimension of the array $work$.

Constraints:

$lwork \geq 1$

$lwork \geq \max(3 * \min(m, n) + \max(m, n), 5 * \min(m, n))$ (for real flavors);

$lwork \geq 2 * \min(m, n) + \max(m, n)$ (for complex flavors).

For good performance, $lwork$ must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* for details.

$rwork$ REAL for `cgesvd`

DOUBLE PRECISION for `zgesvd`

Workspace array, size at least $\max(1, 5 \cdot \min(m, n))$. Used in complex flavors only.

Output Parameters

<code>a</code>	<p>On exit,</p> <p>If <code>jobu = 'O'</code>, <code>a</code> is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors stored columnwise);</p> <p>If <code>jobvt = 'O'</code>, <code>a</code> is overwritten with the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored rowwise);</p> <p>If <code>jobu ≠ 'O'</code> and <code>jobvt ≠ 'O'</code>, the contents of <code>a</code> are destroyed.</p>
<code>s</code>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.</p>
<code>u, vt</code>	<p>REAL for <code>sgesvd</code></p> <p>DOUBLE PRECISION for <code>dgesvd</code></p> <p>COMPLEX for <code>cgesvd</code></p> <p>DOUBLE COMPLEX for <code>zgesvd</code>.</p> <p>Arrays:</p> <p><code>u(ldu,*)</code>; the second dimension of <code>u</code> must be at least $\max(1, m)$ if <code>jobu = 'A'</code>, and at least $\max(1, \min(m, n))$ if <code>jobu = 'S'</code>.</p> <p>If <code>jobu = 'A'</code>, <code>u</code> contains the m-by-m orthogonal/unitary matrix U.</p> <p>If <code>jobu = 'S'</code>, <code>u</code> contains the first $\min(m, n)$ columns of U (the left singular vectors stored column-wise).</p> <p>If <code>jobu = 'N'</code> or <code>'O'</code>, <code>u</code> is not referenced.</p> <p><code>vt(ldvt,*)</code>; the second dimension of <code>vt</code> must be at least $\max(1, n)$.</p> <p>If <code>jobvt = 'A'</code>, <code>vt</code> contains the n-by-n orthogonal/unitary matrix V^T/V^H.</p> <p>If <code>jobvt = 'S'</code>, <code>vt</code> contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).</p> <p>If <code>jobvt = 'N'</code> or <code>'O'</code>, <code>vt</code> is not referenced.</p>
<code>work</code>	<p>On exit, if <code>info = 0</code>, then <code>work(1)</code> returns the required minimal size of <code>lwork</code>.</p> <p>For real flavors:</p> <p>If <code>info > 0</code>, <code>work(2:min(m, n))</code> contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in <code>s</code> (not necessarily sorted). B satisfies $A = u * B * vt$, so it has the same singular values as A, and singular vectors related by <code>u</code> and <code>vt</code>.</p>

<i>rwork</i>	On exit (for complex flavors), if <i>info</i> > 0, <i>rwork</i> (1:min(<i>m</i> , <i>n</i>)-1) contains the unconverged superdiagonal elements of an upper bidiagonal matrix <i>B</i> whose diagonal is in <i>s</i> (not necessarily sorted). <i>B</i> satisfies $A = u*B*vt$, so it has the same singular values as <i>A</i> , and singular vectors related by <i>u</i> and <i>vt</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then if ?bdsqr did not converge, <i>i</i> specifies how many superdiagonals of the intermediate bidiagonal form <i>B</i> did not converge to zero (see the description of the <i>work</i> and <i>rwork</i> parameters for details).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *gesvd* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m</i> , <i>n</i>).
<i>s</i>	Holds the vector of length min(<i>m</i> , <i>n</i>).
<i>u</i>	If present and is a square <i>m</i> -by- <i>m</i> matrix, on exit contains the <i>m</i> -by- <i>m</i> orthogonal/unitary matrix <i>U</i> . Otherwise, if present, on exit contains the first min(<i>m</i> , <i>n</i>) columns of the matrix <i>U</i> (left singular vectors stored column-wise).
<i>vt</i>	If present and is a square <i>n</i> -by- <i>n</i> matrix, on exit contains the <i>n</i> -by- <i>n</i> orthogonal/unitary matrix V^T/V^H . Otherwise, if present, on exit contains the first min(<i>m</i> , <i>n</i>) rows of the matrix V^T/V^H (right singular vectors stored row-wise).
<i>ww</i>	Holds the vector of length min(<i>m</i> , <i>n</i>)-1. <i>ww</i> contains the unconverged superdiagonal elements of an upper bidiagonal matrix <i>B</i> whose diagonal is in <i>s</i> (not necessarily sorted). <i>B</i> satisfies $A = U*B*VT$, so it has the same singular values as <i>A</i> , and singular vectors related by <i>U</i> and <i>VT</i> .
<i>job</i>	Must be either 'N', or 'U', or 'V'. The default value is 'N'. If <i>job</i> = 'U', and <i>u</i> is not present, then <i>u</i> is returned in the array <i>a</i> . If <i>job</i> = 'V', and <i>vt</i> is not present, then <i>vt</i> is returned in the array <i>a</i> .

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

```
call sgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call cgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call zgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call gesdd(a, s [,u] [,vt] [,jobz] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide-and-conquer algorithm. The SVD is written

$A = U \Sigma V^T$ for real routines,

$A = U \Sigma V^H$ for complex routines,

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns $vt = V^T$ (for real flavors) or $vt = V^H$ (for complex flavors), not V .

Input Parameters

jobz CHARACTER*1. Must be 'A', 'S', 'O', or 'N'.
 Specifies options for computing all or part of the matrices U and V .
 If $jobz = 'A'$, all m columns of U and all n rows of V^T or V^H are returned in the arrays u and vt ;
 if $jobz = 'S'$, the first $\min(m, n)$ columns of U and the first $\min(m, n)$ rows of V^T or V^H are returned in the arrays u and vt ;
 if $jobz = 'O'$, then
 if $m \geq n$, the first n columns of U are overwritten in the array a and all rows of V^T or V^H are returned in the array vt ;
 if $m < n$, all columns of U are returned in the array u and the first m rows of V^T or V^H are overwritten in the array a ;

if $jobz = 'N'$, no columns of U or rows of V^T or V^H are computed.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for `sgesdd`
 DOUBLE PRECISION for `dgesdd`
 COMPLEX for `cgesdd`
 DOUBLE COMPLEX for `zgesdd`.

Arrays:
 $a(lda,*)$ is an array containing the m -by- n matrix A .
 The second dimension of a must be at least $\max(1, n)$.
 $work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array a . Must be at least $\max(1, m)$.

$ldu, ldvt$ INTEGER. The leading dimensions of the output arrays u and vt , respectively.

Constraints:
 $ldu \geq 1; ldvt \geq 1$.
 If $jobz = 'S'$ or $'A'$, or $jobz = 'O'$ and $m < n$,
 then $ldu \geq m$;
 If $jobz = 'A'$ or $jobz = 'O'$ and $m \geq n$,
 then $ldvt \geq n$;
 If $jobz = 'S'$, $ldvt \geq \min(m, n)$.

$lwork$ INTEGER.
 The dimension of the array $work$; $lwork \geq 1$.
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the $work(1)$, and no error message related to $lwork$ is issued by `xerbla`.
 See *Application Notes* for the theoretical minimum value of $lwork$ when a workspace query is not performed.

$rwork$ REAL for `cgesdd`
 DOUBLE PRECISION for `zgesdd`
 Workspace array, size at least $\max(1, 7 * \min(m, n))$ if $jobz = 'N'$.
 Otherwise, the dimension of $rwork$ must be at least
 $\max(1, \min(m, n) * \max(5 * \min(m, n) + 7, 2 * \max(m, n) + 2 * \min(m, n) + 1))$.
 This array is used in complex flavors only.

$iwork$ INTEGER. Workspace array, size at least $\max(1, 8 * \min(m, n))$.

Output Parameters

<i>a</i>	<p>On exit:</p> <p>If <i>jobz</i> = 'O', then if $m \geq n$, <i>a</i> is overwritten with the first <i>n</i> columns of <i>U</i> (the left singular vectors, stored columnwise). If $m < n$, <i>a</i> is overwritten with the first <i>m</i> rows of V^T (the right singular vectors, stored rowwise);</p> <p>If <i>jobz</i> ≠ 'O', the contents of <i>a</i> are destroyed.</p>
<i>s</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, size at least $\max(1, \min(m, n))$. Contains the singular values of <i>A</i> sorted so that $s(i) \geq s(i+1)$.</p>
<i>u, vt</i>	<p>REAL for sgesdd</p> <p>DOUBLE PRECISION for dgesdd</p> <p>COMPLEX for cgesdd</p> <p>DOUBLE COMPLEX for zgesdd.</p> <p>Arrays:</p> <p><i>u(ldu,*)</i>; the second dimension of <i>u</i> must be at least $\max(1, m)$ if <i>jobz</i> = 'A' or <i>jobz</i> = 'O' and $m < n$.</p> <p>If <i>jobz</i> = 'S', the second dimension of <i>u</i> must be at least $\max(1, \min(m, n))$.</p> <p>If <i>jobz</i> = 'A' or <i>jobz</i> = 'O' and $m < n$, <i>u</i> contains the <i>m</i>-by-<i>m</i> orthogonal/unitary matrix <i>U</i>.</p> <p>If <i>jobz</i> = 'S', <i>u</i> contains the first $\min(m, n)$ columns of <i>U</i> (the left singular vectors, stored columnwise).</p> <p>If <i>jobz</i> = 'O' and $m \geq n$, or <i>jobz</i> = 'N', <i>u</i> is not referenced.</p> <p><i>vt(ldvt,*)</i>; the second dimension of <i>vt</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'A' or <i>jobz</i> = 'O' and $m \geq n$, <i>vt</i> contains the <i>n</i>-by-<i>n</i> orthogonal/unitary matrix V^T.</p> <p>If <i>jobz</i> = 'S', <i>vt</i> contains the first $\min(m, n)$ rows of V^T (the right singular vectors, stored rowwise).</p> <p>If <i>jobz</i> = 'O' and $m < n$, or <i>jobz</i> = 'N', <i>vt</i> is not referenced.</p>
<i>work(l)</i>	<p>On exit, if <i>info</i> = 0, then <i>work(l)</i> returns the optimal size of <i>lwork</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, then ?bdsdc did not converge, updating process failed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gesdd` interface are the following:

<code>a</code>	Holds the matrix A of size (m, n) .
<code>s</code>	Holds the vector of length $\min(m, n)$.
<code>u</code>	Holds the matrix U of size <ul style="list-style-type: none"> (m, m) if <code>jobz='A'</code> or <code>jobz='O'</code> and $m < n$ $(m, \min(m, n))$ if <code>jobz='S'</code> <code>u</code> is not referenced if <code>jobz</code> is not supplied or if <code>jobz='N'</code> or <code>jobz='O'</code> and $m \geq n$.
<code>vt</code>	Holds the matrix VT of size <ul style="list-style-type: none"> (n, n) if <code>jobz='A'</code> or <code>jobz='O'</code> and $m \geq n$ $(\min(m, n), n)$ if <code>jobz='S'</code> <code>vt</code> is not referenced if <code>jobz</code> is not supplied or if <code>jobz='N'</code> or <code>jobz='O'</code> and $m < n$.
<code>job</code>	Must be 'N', 'A', 'S', or 'O'. The default value is 'N'.

Application Notes

The theoretical minimum value for `lwork` depends on the flavor of the routine.

For real flavors:

If `jobz = 'N'`, `lwork`= $3 \cdot \min(m, n) + \max(\max(m, n), 6 \cdot \min(m, n))$;

If `jobz = 'O'`, `lwork`= $3 \cdot (\min(m, n))^2 + \max(\max(m, n), 5 \cdot (\min(m, n))^2 + 4 \cdot \min(m, n))$;

If `jobz = 'S'` or 'A', `lwork`= $\min(m, n) \cdot (6 + 4 \cdot \min(m, n)) + \max(m, n)$;

For complex flavors:

If `jobz = 'N'`, `lwork`= $2 \cdot \min(m, n) + \max(m, n)$;

If `jobz = 'O'`, `lwork`= $2 \cdot (\min(m, n))^2 + \max(m, n) + 2 \cdot \min(m, n)$;

If `jobz = 'S'` or 'A', `lwork`= $(\min(m, n))^2 + \max(m, n) + 2 \cdot \min(m, n)$;

The optimal value of `lwork` returned by a workspace query generally provides better performance than the theoretical minimum value. The value of `lwork` returned by a workspace query is generally larger than the theoretical minimum value, but for very small matrices it can be smaller. The absolute minimum value of `lwork` is the minimum of the workspace query result and the theoretical minimum.

If you set `lwork` to a value less than the absolute minimum value and not equal to -1, the routine returns immediately with an error exit and does not provide information on the recommended workspace size.

?gejsv

Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.

Syntax

```
call sgejsv(joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv,
work, lwork, iwork, info)
```

```
call dgejsv(joba, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv,
work, lwork, iwork, info)
```

Include Files

- `mkl.fi`

Description

The routine computes the singular value decomposition (SVD) of a real m -by- n matrix A , where $m \geq n$.

The SVD is written as

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix which is zero except for its n diagonal elements, U is an m -by- n (or m -by- m) orthonormal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array `sva`.

The `?gejsv` routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The routine implements a preconditioned Jacobi SVD algorithm. It uses `?geqp3`, `?geqrf`, and `?gelqf` as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix A with the structure $A = D1 * C * D2$, where $D1$, $D2$ are arbitrarily ill-conditioned diagonal matrices and C is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of C , and independent of $D1$, $D2$. Such higher accuracy is not completely understood theoretically, but it works well in practice.

If A can be written as $A = B * D$, with well-conditioned B and some diagonal D , then the high accuracy is guaranteed, both theoretically and in software independent of D . For more details see [[Drmac08-1](#)], [[Drmac08-2](#)].

The computational range for the singular values can be the full range (`UNDERFLOW,OVERFLOW`), provided that the machine arithmetic and the BLAS and LAPACK routines called by `?gejsv` are implemented to work in that range. If that is not the case, the restriction for safe computation with the singular values in the range of normalized IEEE numbers is that the spectral condition number $\kappa(A) = \sigma_{\max}(A) / \sigma_{\min}(A)$ does not overflow. This code (`?gejsv`) is best used in this restricted range, meaning that singular values of magnitude below $\|A\|_2 / \text{slamch}('O')$ (for single precision) or $\|A\|_2 / \text{dlamch}('O')$ (for double precision) are returned as zeros. See `jobr` for details on this.

This implementation is slower than the one described in [[Drmac08-1](#)], [[Drmac08-2](#)] due to replacement of some non-LAPACK components, and because the choice of some tuning parameters in the iterative part (`?gesvj`) is left to the implementer on a particular machine.

The rank revealing QR factorization (in this code: `?geqp3`) should be implemented as in [[Drmac08-3](#)].

If m is much larger than n , it is obvious that the initial QRF with column pivoting can be preprocessed by the QRF without pivoting. That well known trick is not used in `?gejsv` because in some cases heavy row weighting can be treated with complete pivoting. The overhead in cases m much larger than n is then only due to pivoting, but the benefits in accuracy have prevailed. You can incorporate this extra QRF step easily and also improve data movement (matrix transpose, matrix copy, matrix transposed copy) - this implementation of `?gejsv` uses only the simplest, naive data movement.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-

Optimization Notice

dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

joba

CHARACTER*1. Must be 'C', 'E', 'F', 'G', 'A', or 'R'.

Specifies the level of accuracy:

If *joba* = 'C', high relative accuracy is achieved if $A = B \cdot D$ with well-conditioned B and arbitrary diagonal matrix D . The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of B , and the procedure aims at the best theoretical accuracy. The relative error $\max_{i=1:N} |\delta \sigma_i| / \sigma_i$ is bounded by $f(M,N) \cdot \epsilon \cdot \text{cond}(B)$, independent of D . The input matrix is preprocessed with the QRF with column pivoting. This initial preprocessing and preconditioning by a rank revealing QR factorization is common for all values of *joba*. Additional actions are specified as follows:

If *joba* = 'E', computation as with 'C' with an additional estimate of the condition number of B . It provides a realistic error bound.

If *joba* = 'F', accuracy higher than in the 'C' option is achieved, if $A = D1 \cdot C \cdot D2$ with ill-conditioned diagonal scalings $D1$, $D2$, and a well-conditioned matrix C . This option is advisable, if the structure of the input matrix is not known and relative accuracy is desirable. The input matrix A is preprocessed with QR factorization with full (row and column) pivoting.

If *joba* = 'G', computation as with 'F' with an additional estimate of the condition number of B , where $A = B \cdot D$. If A has heavily weighted rows, using this condition number gives too pessimistic error bound.

If *joba* = 'A', small singular values are the noise and the matrix is treated as numerically rank deficient. The error in the computed singular values is bounded by $f(m,n) \cdot \epsilon \cdot \|A\|$. The computed SVD $A = U \cdot S \cdot V^T$ restores A up to $f(m,n) \cdot \epsilon \cdot \|A\|$. This enables the procedure to set all singular values below $n \cdot \epsilon \cdot \|A\|$ to zero.

If *joba* = 'R', the procedure is similar to the 'A' option. Rank revealing property of the initial QR factorization is used to reveal (using triangular factor) a gap $\sigma_{r+1} < \epsilon \cdot \sigma_r$, in which case the numerical rank is declared to be r . The SVD is computed with absolute error bounds, but more accurately than with 'A'.

jobu

CHARACTER*1. Must be 'U', 'F', 'W', or 'N'.

Specifies whether to compute the columns of the matrix U :

If *jobu* = 'U', n columns of U are returned in the array u

If *jobu* = 'F', a full set of m left singular vectors is returned in the array u .

If *jobu* = 'W', u may be used as workspace of length $m \cdot n$. See the description of u .

If $jobu = 'N'$, u is not computed.

jobv

CHARACTER*1. Must be 'V', 'J', 'W', or 'N'.

Specifies whether to compute the matrix V :

If $jobv = 'V'$, n columns of V are returned in the array v ; Jacobi rotations are not explicitly accumulated.

If $jobv = 'J'$, n columns of V are returned in the array v but they are computed as the product of Jacobi rotations. This option is allowed only if $jobu \neq 'N'$

If $jobv = 'W'$, v may be used as workspace of length $n*n$. See the description of v .

If $jobv = 'N'$, v is not computed.

jobr

CHARACTER*1. Must be 'N' or 'R'.

Specifies the range for the singular values. If small positive singular values are outside the specified range, they may be set to zero. If A is scaled so that the largest singular value of the scaled matrix is around $\sqrt{\text{big}}$, $\text{big} = \text{?lamch}('O')$, the function can remove columns of A whose norm in the scaled matrix is less than $\sqrt{\text{?lamch}('S')}$ (for $jobr = 'R'$), or less than $\text{small} = \text{?lamch}('S')/\text{?lamch}('E')$.

If $jobr = 'N'$, the function does not remove small columns of the scaled matrix. This option assumes that BLAS and QR factorizations and triangular solvers are implemented to work in that range. If the condition of A is greater than big , use ?gesvj .

If $jobr = 'R'$, restricted range for singular values of the scaled matrix A is $[\sqrt{\text{?lamch}('S')}, \sqrt{\text{big}}]$, roughly as described above. This option is recommended.

For computing the singular values in the full range $[\text{?lamch}('S'), \text{big}]$, use ?gesvj .

jobt

CHARACTER*1. Must be 'T' or 'N'.

If the matrix is square, the procedure may determine to use a transposed A if A^T seems to be better with respect to convergence. If the matrix is not square, $jobt$ is ignored. This is subject to changes in the future.

The decision is based on two values of entropy over the adjoint orbit of $A^T * A$. See the descriptions of $work(6)$ and $work(7)$.

If $jobt = 'T'$, the function performs transposition if the entropy test indicates possibly faster convergence of the Jacobi process, if A is taken as input. If A is replaced with A^T , the row pivoting is included automatically.

If $jobt = 'N'$, the functions attempts no speculations. This option can be used to compute only the singular values, or the full SVD (u , σ , and v). For only one set of singular vectors (u or v), the caller should provide both u and v , as one of the arrays is used as workspace if the matrix A is transposed. The implementer can easily remove this constraint and make the code more complicated. See the descriptions of u and v .

jobp

CHARACTER*1. Must be 'P' or 'N'.

Enables structured perturbations of denormalized numbers. This option should be active if the denormals are poorly implemented, causing slow computation, especially in cases of fast convergence. For details, see [Drmac08-1], [Drmac08-2]. For simplicity, such perturbations are included only when the full SVD or only the singular values are requested. You can add the perturbation for the cases of computing one set of singular vectors.

If $jobp = 'P'$, the function introduces perturbation.

If $jobp = 'N'$, the function introduces no perturbation.

m	INTEGER. The number of rows of the input matrix A ; $m \geq 0$.
n	INTEGER. The number of columns in the input matrix A ; $m \geq n \geq 0$.
$a, work, sva, u, v$	REAL for <code>sgejsv</code> DOUBLE PRECISION for <code>dgejsv</code> . Array $a(lda,*)$ is an array containing the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$. sva is a workspace array, its size is n . u is a workspace array, its size is $(ldu,*)$; the second dimension of u must be at least $\max(1, n)$. v is a workspace array, its size is $(ldv,*)$; the second dimension of u must be at least $\max(1, n)$.
lda	INTEGER. The leading dimension of the array a . Must be at least $\max(1, m)$.
ldu	INTEGER. The leading dimension of the array u ; $ldu \geq 1$. $jobu = 'U'$ or $'F'$ or $'W'$, $ldu \geq m$ for column major layout.
ldv	INTEGER. The leading dimension of the array v ; $ldv \geq 1$. $jobv = 'V'$ or $'J'$ or $'W'$, $ldv \geq n$.
$lwork$	INTEGER. Length of $work$ to confirm proper allocation of work space. $lwork$ depends on the task performed: If only σ is needed ($jobu = 'N'$, $jobv = 'N'$) and <ul style="list-style-type: none"> ... no scaled condition estimate is required, then $lwork \geq \max(2*m+n, 4*n+1, 7)$. This is the minimal requirement. For optimal performance (blocked code) the optimal value is $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, 7)$. Here nb is the optimal block size for <code>?geqp3/?geqrf</code>. In general, the optimal length $lwork$ is computed as $lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(sgeqrf), 7)$ for <code>sgejsv</code> $lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dgeqrf), 7)$ for <code>dgejsv</code>

- ... an estimate of the scaled condition number of A is required ($joba = 'E', 'G'$). In this case, $lwork$ is the maximum of the above and $n*n + 4*n$, that is, $lwork \geq \max(2*m+n, n*n+4*n, 7)$. For optimal performance (blocked code) the optimal value is $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, n*n+4*n, 7)$.

In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(sgeqrf), n+n*n+lwork(spocon), 7)$ for $sgejsv$

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dgeqrf), n+n*n+lwork(dpocon), 7)$ for $dgejsv$

If σ and the right singular vectors are needed ($jobv = 'V'$),

- the minimal requirement is $lwork \geq \max(2*m+n, 4*n+1, 7)$.
- for optimal performance, $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, 7)$, where nb is the optimal block size for $?geqp3, ?geqrf, ?gelqf, ?ormlq$. In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(spocon), n+lwork(sgelqf), 2*n+lwork(sgeqrf), n+lwork(sormlq))$ for $sgejsv$

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dpocon), n+lwork(dgelqf), 2*n+lwork(dgeqrf), n+lwork(dormlq))$ for $dgejsv$

If σ and the left singular vectors are needed

- the minimal requirement is $lwork \geq \max(2*n+m, 4*n+1, 7)$.
- for optimal performance,
 - if $jobu = 'U' :: lwork \geq \max(2*m+n, 3*n+(n+1)*nb, 7)$,
 - if $jobu = 'F' :: lwork \geq \max(2*m+n, 3*n+(n+1)*nb, n*m*nb, 7)$,

where nb is the optimal block size for $?geqp3, ?geqrf, ?ormlq$. In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(spocon), 2*n+lwork(sgeqrf), n+lwork(sormlq))$ for $sgejsv$

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dpocon), 2*n+lwork(dgeqrf), n+lwork(dormlq))$ for $dgejsv$

Here $lwork(?ormlq)$ equals $n*nb$ (for $jobu = 'U'$) or $m*nb$ (for $jobu = 'F'$)

If full SVD is needed ($jobu = 'U'$ or $'F'$) and

- if $jobv = 'V'$,
 - the minimal requirement is $lwork \geq \max(2*m+n, 6*n+2*n*n)$
- if $jobv = 'J'$,
 - the minimal requirement is $lwork \geq \max(2*m+n, 4*n+n*n, 2*n+n*n+6)$
- For optimal performance, $lwork$ should be additionally larger than $n + m*nb$, where nb is the optimal block size for $?ormlq$.

iwork INTEGER. Workspace array, size $\max(3, m+3*n)$.

Output Parameters

sva

On exit:

For $work(1)/work(2) = one$: the singular values of A . During the computation *sva* contains Euclidean column norms of the iterated matrices in the array *a*.

For $work(1) \neq work(2)$: the singular values of A are $(work(1)/work(2)) * sva(1:n)$. This factored form is used if $\sigma_{max}(A)$ overflows or if small singular values have been saved from underflow by scaling the input matrix A .

$jobr = 'R'$, some of the singular values may be returned as exact zeros obtained by 'setting to zero' because they are below the numerical rank threshold or are denormalized numbers.

u

On exit:

If $jobu = 'U'$, contains the m -by- n matrix of the left singular vectors.

If $jobu = 'F'$, contains the m -by- m matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of the range of A .

If $jobu = 'W'$ and $jobv = 'V'$, $jobt = 'T'$, and $m = n$, then *u* is used as workspace if the procedure replaces A with A^T . In that case, *v* is computed in *u* as left singular vectors of A^T and copied back to the *v* array. This 'W' option is just a reminder to the caller that in this case *u* is reserved as workspace of length $n*n$.

If $jobu = 'N'$, *u* is not referenced.

v

On exit:

If $jobv = 'V'$ or $'J'$, contains the n -by- n matrix of the right singular vectors.

If $jobv = 'W'$ and $jobu = 'U'$, $jobt = 'T'$, and $m = n$, then *v* is used as workspace if the procedure replaces A with A^T . In that case, *u* is computed in *v* as right singular vectors of A^T and copied back to the *u* array. This 'W' option is just a reminder to the caller that in this case *v* is reserved as workspace of length $n*n$.

If $jobv = 'N'$, *v* is not referenced.

work

On exit,

$work(1) = scale = work(2)/work(1)$ is the scaling factor such that $scale*sva(1:n)$ are the computed singular values of A . See the description of *sva()*.

$work(2) =$ see the description of *work(1)*.

$work(3) = sconda$ is an estimate for the condition number of column equilibrated A . If $joba = 'E'$ or $'G'$, $sconda$ is an estimate of $\sqrt{(|R^{*t} * R|^{(-1)} ||_1)}$. It is computed using *?pocon*. It holds $n^{*(-1/4)} * sconda \leq ||R^{*(-1)} ||_2 \leq n^{*(1/4)} * sconda$, where R

is the triangular factor from the QRF of A . However, if R is truncated and the numerical rank is determined to be strictly smaller than n , *sconda* is returned as -1, indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

work(4) = an estimate of the scaled condition number of the triangular factor in the first QR factorization.

work(5) = an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if *jobt* = 'T'. They are provided for a user who is familiar with the details of the method.

work(6) = the entropy of $A^{**t}A$: : this is the Shannon entropy of $\text{diag}(A^{**t}A) / \text{Trace}(A^{**t}A)$ taken as point in the probability simplex.

work(7) = the entropy of A^*A^{**t} .

iwork

INTEGER. On exit,

iwork(1) = the numerical rank determined after the initial QR factorization with pivoting. See the descriptions of *joba* and *jobr*.

iwork(2) = the number of the computed nonzero singular value.

iwork(3) = if nonzero, a warning message. If *iwork(3)*=1, some of the column norms of A were denormalized floats. The requested high accuracy is not warranted by the data.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number of sweeps. The computed values may be inaccurate.

See Also

[?geqp3](#)

[?geqrf](#)

[?gelqf](#)

[?gesvj](#)

[?lamch](#)

[?pocon](#)

[?ormlq](#)

[?gesvj](#)

Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

Syntax

```
call sgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
```

```
call dgesvj(joba, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
```

Include Files

- `mk1.fi`

Description

The routine computes the singular value decomposition (SVD) of a real m -by- n matrix A , where $m \geq n$.

The SVD of A is written as

$$A = U \Sigma V^T,$$

where Σ is an m -by- n diagonal matrix, U is an m -by- n orthonormal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array sva .

The `?gesvj` routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The n -by- n orthogonal matrix V is obtained as a product of Jacobi plane rotations. The rotations are implemented as fast scaled rotations of Anda and Park [[AndaPark94](#)]. In the case of underflow of the Jacobi angle, a modified Jacobi transformation of Drmac ([[Drmac08-4](#)]) is used. Pivot strategy uses column interchanges of de Rijk ([[deRijk98](#)]). The relative accuracy of the computed singular values and the accuracy of the computed singular vectors (in angle metric) is as guaranteed by the theory of Demmel and Veselic [[Demmel92](#)]. The condition number that determines the accuracy in the full rank case is essentially

$$\left(\min_i d_{ii} \right) \cdot \kappa(A \cdot D)$$

where $\kappa(\cdot)$ is the spectral condition number. The best performance of this Jacobi SVD procedure is achieved if used in an accelerated version of Drmac and Veselic [[Drmac08-1](#)], [[Drmac08-2](#)].

The computational range for the nonzero singular values is the machine number interval (`UNDERFLOW,OVERFLOW`). In extreme cases, even denormalized singular values can be computed with the corresponding gradual loss of accurate digit.

Input Parameters

<code>joba</code>	<p>CHARACTER*1. Must be 'L', 'U' or 'G'.</p> <p>Specifies the structure of A:</p> <p>If <code>joba</code> = 'L', the input matrix A is lower triangular.</p> <p>If <code>joba</code> = 'U', the input matrix A is upper triangular.</p> <p>If <code>joba</code> = 'G', the input matrix A is a general m-by-n, $m \geq n$.</p>
<code>jobu</code>	<p>CHARACTER*1. Must be 'U', 'C' or 'N'.</p> <p>Specifies whether to compute the left singular vectors (columns of U):</p> <p>If <code>jobu</code> = 'U', the left singular vectors corresponding to the nonzero singular values are computed and returned in the leading columns of A. See more details in the description of <code>a</code>. The default numerical orthogonality threshold is set to approximately <code>TOL=CTOL*EPS</code>, <code>CTOL=sqrt(m)</code>, <code>EPS = ?lamch('E')</code></p> <p>If <code>jobu</code> = 'C', analogous to <code>jobu</code> = 'U', except that you can control the level of numerical orthogonality of the computed left singular vectors. <code>TOL</code> can be set to <code>TOL=CTOL*EPS</code>, where <code>CTOL</code> is given on input in the array</p>

work. No CTOL smaller than ONE is allowed. CTOL greater than $1 / \text{EPS}$ is meaningless. The option 'C' can be used if $m \cdot \text{EPS}$ is satisfactory orthogonality of the computed left singular vectors, so $\text{CTOL} = m$ could save a few sweeps of Jacobi rotations. See the descriptions of *a* and *work(1)*.

If *jobu* = 'N', *u* is not computed. However, see the description of *a*.

jobv

CHARACTER*1. Must be 'V', 'A' or 'N'.

Specifies whether to compute the right singular vectors, that is, the matrix *V*:

If *jobv* = 'V', the matrix *V* is computed and returned in the array *v*.

If *jobv* = 'A', the Jacobi rotations are applied to the *mv*-by-*n* array *v*. In other words, the right singular vector matrix *V* is not computed explicitly, instead it is applied to an *mv*-by-*n* matrix initially stored in the first *mv* rows of *V*.

If *jobv* = 'N', the matrix *V* is not computed and the array *v* is not referenced.

m

INTEGER. The number of rows of the input matrix *A*.

$1/\text{slamch}('E') > m \geq 0$ for *sgesvj*.

$1/\text{dlamch}('E') > m \geq 0$ for *dgesvj*.

n

INTEGER. The number of columns in the input matrix *A*; $m \geq n \geq 0$.

a, work, sva, v

REAL for *sgesvj*

DOUBLE PRECISION for *dgesvj*.

Array *a(lda,*)* is an array containing the *m*-by-*n* matrix *A*.

The second dimension of *a* is $\max(1, n)$.

work is a workspace array, its dimension $\max(4, m+n)$.

If *jobu* = 'C', *work(1)*=CTOL, where CTOL defines the threshold for convergence. The process stops if all columns of *A* are mutually orthogonal up to $\text{CTOL} \cdot \text{EPS}$, $\text{EPS} = ?\text{lamch}('E')$. It is required that $\text{CTOL} \geq 1$, that is, it is not allowed to force the routine to obtain orthogonality below ϵ .

sva is a workspace array, its dimension is *n*.

Array *v* is a workspace array, its dimension is (*ldv,**); the second dimension of *u* must be at least $\max(1, n)$.

lda

INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, m)$.

mv

INTEGER.

If *jobv* = 'A', the product of Jacobi rotations in *?gesvj* is applied to the first *mv* rows of *v*. See the description of *jobv*. $0 \leq mv \leq ldv$.

ldv

INTEGER. The leading dimension of the array *v*; $ldv \geq 1$.

jobv = 'V', $ldv \geq \max(1, n)$.

jobv = 'A', $ldv \geq \max(1, mv)$.

lwork

INTEGER.

Length of *work*, $work \geq \max(6, m+n)$.

Output Parameters

a

On exit:

If *jobu* = 'U' or *jobu* = 'C':

- if *info* = 0, the leading columns of *A* contain left singular vectors corresponding to the computed singular values of *a* that are above the underflow threshold ?lamch('S') , that is, non-zero singular values. The number of the computed non-zero singular values is returned in *work*(2). Also see the descriptions of *sva* and *work*. The computed columns of *u* are mutually numerically orthogonal up to approximately $\text{TOL} = \text{sqrt}(m) * \text{EPS}$ (default); or $\text{TOL} = \text{CTOL} * \text{EPS}$ *jobu* = 'C', see the description of *jobu*.
- if *info* > 0, the procedure *?gesvj* did not converge in the given number of iterations (sweeps). In that case, the computed columns of *u* may not be orthogonal up to *TOL*. The output *u* (stored in *a*), *sigma* (given by the computed singular values in *sva*(1:n)) and *v* is still a decomposition of the input matrix *A* in the sense that the residual $\| |A - \text{scale} * U * \text{sigma} * V^T | \|_2 / \| |A | \|_2$ (where *scale* = *stat*[0]) is small.

If *jobu* = 'N':

- if *info* = 0, note that the left singular vectors are 'for free' in the one-sided Jacobi SVD algorithm. However, if only the singular values are needed, the level of numerical orthogonality of *u* is not an issue and iterations are stopped when the columns of the iterated matrix are numerically orthogonal up to approximately $m * \text{EPS}$. Thus, on exit, *a* contains the columns of *u* scaled with the corresponding singular values.
- if *info* > 0, the procedure *?gesvj* did not converge in the given number of iterations (sweeps).

sva

On exit:

If *info* = 0, depending on the value *scale* = *work*(1), where *scale* is the scaling factor:

- if *scale* = 1, *sva*(1:n) contains the computed singular values of *a*.
During the computation, *sva* contains the Euclidean column norms of the iterated matrices in the array *a*.
- if *scale* ≠ 1, the singular values of *a* are *scale***sva*(1:n), and this factored representation is due to the fact that some of the singular values of *a* might underflow or overflow.

If *info* > 0, the procedure *?gesvj* did not converge in the given number of iterations (sweeps) and *scale***sva*(1:n) may not be accurate.*v*

On exit:

If *jobv* = 'V', contains the *n*-by-*n* matrix of the right singular vectors.If *jobv* = 'A', then *v* contains the product of the computed right singular vector matrix and the initial matrix in the array *v*.

If `jobv = 'N'`, `v` is not referenced.

`work`

On exit,

`work(1) = scale` is the scaling factor such that `scale*sva(1:n)` are the computed singular values of `A`. See the description of `sva()`.

`work(2)` is the number of the computed nonzero singular value.

`work(3)` is the number of the computed singular values that are larger than the underflow threshold.

`work(4)` is the number of sweeps of Jacobi rotations needed for numerical convergence.

`work(5) = max_{i≠j} |COS(A(:,i),A(:,j))|` in the last sweep. This is useful information in cases when `?gesvj` did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.

`work(6)` is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful in a post festum analysis.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info > 0`, the function did not converge in the maximal number (30) of sweeps. The output may still be useful. See the description of `work`.

See Also

[?lamch](#)

[?ggsvd](#)

Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Syntax

```
call sggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, iwork, info)
```

```
call dggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, iwork, info)
```

```
call cggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, rwork, iwork, info)
```

```
call zggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, rwork, iwork, info)
```

```
call ggsvd(a, b, alpha, beta [, k] [,l] [,u] [,v] [,q] [,iwork] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

$$D_2 = \begin{matrix} & & k & m - k & k + l - m \\ m - k & \begin{pmatrix} 0 & S & 0 \end{pmatrix} \\ k + l - m & \begin{pmatrix} 0 & 0 & I \end{pmatrix} \\ p - l & \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

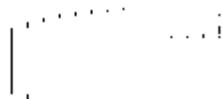
$$(0 \ R) = \begin{matrix} & & n - k - l & k & m - k & k + l - m \\ & k & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \end{pmatrix} \\ m - k & \begin{pmatrix} 0 & 0 & R_{22} & R_{23} \end{pmatrix} \\ k + l - m & \begin{pmatrix} 0 & 0 & 0 & R_{33} \end{pmatrix} \end{matrix}$$

where

$$C = \text{diag}(\alpha(K+1), \dots, \alpha(m)),$$

$$S = \text{diag}(\beta(K+1), \dots, \beta(m)),$$

$$C_2 + S_2 = I$$



On exit, C is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored in $b(m-k+1:l, n+m-k-l+1:n)$.

The routine computes C , S , R , and optionally the orthogonal/unitary transformation matrices U , V and Q .

In particular, if B is an n -by- n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of $A*B^{-1}$:

$$A*B^{-1} = U*(D_1*D_2^{-1})*V'$$

If (A', B') has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A'*A*x = \lambda*B'*B*x.$$

Input Parameters

- jobu* CHARACTER*1. Must be 'U' or 'N'.
If *jobu* = 'U', orthogonal/unitary matrix U is computed.
If *jobu* = 'N', U is not computed.
- jobv* CHARACTER*1. Must be 'V' or 'N'.
If *jobv* = 'V', orthogonal/unitary matrix V is computed.
If *jobv* = 'N', V is not computed.
- jobq* CHARACTER*1. Must be 'Q' or 'N'.
If *jobq* = 'Q', orthogonal/unitary matrix Q is computed.
If *jobq* = 'N', Q is not computed.

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>p</i>	INTEGER. The number of rows of the matrix <i>B</i> ($p \geq 0$).
<i>a, b, work</i>	REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: <i>a(lda,*)</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b(l db,*)</i> contains the <i>p</i> -by- <i>n</i> matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work(*)</i> is a workspace array. The dimension of <i>work</i> must be at least $\max(3n, m, p)+n$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, p)$.
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> . $ldu \geq \max(1, m)$ if <i>jobu</i> = 'U'; $ldu \geq 1$ otherwise.
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> . $ldv \geq \max(1, p)$ if <i>jobv</i> = 'V'; $ldv \geq 1$ otherwise.
<i>ldq</i>	INTEGER. The leading dimension of the array <i>q</i> . $ldq \geq \max(1, n)$ if <i>jobq</i> = 'Q'; $ldq \geq 1$ otherwise.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, n)$.
<i>rwork</i>	REAL for cggsvd DOUBLE PRECISION for zggsvd. Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of the subblocks. The sum $k+l$ is equal to the effective numerical rank of $(A', B)'$.
<i>a</i>	On exit, <i>a</i> contains the triangular matrix <i>R</i> or part of <i>R</i> .
<i>b</i>	On exit, <i>b</i> contains part of the triangular matrix <i>R</i> if $m-k-1 < 0$.
<i>alpha, beta</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

Arrays, size at least $\max(1, n)$ each.

Contain the generalized singular value pairs of A and B :

$alpha(1:k) = 1,$

$beta(1:k) = 0,$

and if $m-k-l \geq 0,$

$alpha(k+1:k+l) = C,$

$beta(k+1:k+l) = S,$

or if $m-k-l < 0,$

$alpha(k+1:m) = C, alpha(m+1:k+l) = 0$

$beta(k+1:m) = S, beta(m+1:k+l) = 1$

and

$alpha(k+l+1:n) = 0$

$beta(k+l+1:n) = 0.$

u, v, q

REAL for sggsvd

DOUBLE PRECISION for dggsvd

COMPLEX for cggsvd

DOUBLE COMPLEX for zggsvd.

Arrays:

$u(ldu,*)$; the second dimension of u must be at least $\max(1, m)$.

If $jobu = 'U'$, u contains the m -by- m orthogonal/unitary matrix U .

If $jobu = 'N'$, u is not referenced.

$v(ldv,*)$; the second dimension of v must be at least $\max(1, p)$.

If $jobv = 'V'$, v contains the p -by- p orthogonal/unitary matrix V .

If $jobv = 'N'$, v is not referenced.

$q(ldq,*)$; the second dimension of q must be at least $\max(1, n)$.

If $jobq = 'Q'$, q contains the n -by- n orthogonal/unitary matrix Q .

If $jobq = 'N'$, q is not referenced.

$iwork$

On exit, $iwork$ stores the sorting information.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the Jacobi-type procedure failed to converge. For further details, see subroutine [tgsja](#).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ggsvd` interface are the following:

<code>a</code>	Holds the matrix A of size (m, n) .
<code>b</code>	Holds the matrix B of size (p, n) .
<code>alpha</code>	Holds the vector of length n .
<code>beta</code>	Holds the vector of length n .
<code>u</code>	Holds the matrix U of size (m, m) .
<code>v</code>	Holds the matrix V of size (p, p) .
<code>q</code>	Holds the matrix Q of size (n, n) .
<code>iwork</code>	Holds the vector of length n .
<code>jobu</code>	Restored based on the presence of the argument u as follows: <code>jobu</code> = 'U', if u is present, <code>jobu</code> = 'N', if u is omitted.
<code>jobv</code>	Restored based on the presence of the argument v as follows: <code>jobz</code> = 'V', if v is present, <code>jobz</code> = 'N', if v is omitted.
<code>jobq</code>	Restored based on the presence of the argument q as follows: <code>jobz</code> = 'Q', if q is present, <code>jobz</code> = 'N', if q is omitted.

Cosine-Sine Decomposition

This section describes LAPACK driver routines for computing the *cosine-sine decomposition* (CS decomposition). You can also call the corresponding computational routines to perform the same task.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Driver Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines (FORTRAN 77 interface) that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of a block-partitioned orthogonal matrix	<code>orcsd uncsd</code> <code>orcsd2by1 uncsd2by1</code>	
Compute the CS decomposition of a block-partitioned unitary matrix		<code>orcsd uncsd</code>

See Also

[Cosine-Sine Decomposition](#)

?orcsd/?uncsd

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
call sorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, iwork, info )
```

```
call dorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, iwork, info )
```

```
call cuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, rwork, lrwork, iwork, info )
```

```
call zuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, rwork, lrwork, iwork, info )
```

```
call orcsd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )
```

```
call uncscd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orcsd/?uncsd compute the CS decomposition of an m -by- m partitioned orthogonal matrix X :

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} u_1 & | \\ & | & u_2 \end{pmatrix} \begin{pmatrix} I & 0 & 0 & | & 0 & 0 & 0 \\ 0 & C & 0 & | & 0 & -S & 0 \\ 0 & 0 & 0 & | & 0 & 0 & -I \\ 0 & 0 & 0 & | & I & 0 & 0 \\ 0 & S & 0 & | & 0 & C & 0 \\ 0 & 0 & I & | & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 & | \\ & | & v_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} u_1 & | \\ & | & u_2 \end{pmatrix} \begin{pmatrix} I & 0 & 0 & | & 0 & 0 & 0 \\ 0 & C & 0 & | & 0 & -S & 0 \\ 0 & 0 & 0 & | & 0 & 0 & -I \\ 0 & 0 & 0 & | & I & 0 & 0 \\ 0 & S & 0 & | & 0 & C & 0 \\ 0 & 0 & I & | & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 & | \\ & | & v_2 \end{pmatrix}^H$$

x_{11} is p -by- q . The orthogonal/unitary matrices u_1 , u_2 , v_1 , and v_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

<i>jobu1</i>	CHARACTER. If equals <i>Y</i> , then u_1 is computed. Otherwise, u_1 is not computed.
<i>jobu2</i>	CHARACTER. If equals <i>Y</i> , then u_2 is computed. Otherwise, u_2 is not computed.
<i>jobv1t</i>	CHARACTER. If equals <i>Y</i> , then v_1^t is computed. Otherwise, v_1^t is not computed.
<i>jobv2t</i>	CHARACTER. If equals <i>Y</i> , then v_2^t is computed. Otherwise, v_2^t is not computed.
<i>trans</i>	CHARACTER = 'T': $X, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order. otherwise $X, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<i>signs</i>	CHARACTER = 'O': The lower-left block is made nonpositive (the "other" convention). otherwise The upper-right block is made nonpositive (the "default" convention).
<i>m</i>	INTEGER. The number of rows and columns of the matrix X .
<i>p</i>	INTEGER. The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.
<i>q</i>	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
<i>x11, x12, x21, x22</i>	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd DOUBLE COMPLEX for zuncsd Arrays of size <i>x11</i> ($ldx11, q$), <i>x12</i> ($ldx12, m - q$), <i>x21</i> ($ldx21, q$), and <i>x22</i> ($ldx22, m - q$). Contain the parts of the orthogonal/unitary matrix whose CSD is desired.
<i>ldx11, ldx12, ldx21, ldx22</i>	INTEGER. The leading dimensions of the parts of array X . $ldx11 \geq \max(1, p)$, $ldx12 \geq \max(1, p)$, $ldx21 \geq \max(1, m - p)$, $ldx22 \geq \max(1, m - p)$.
<i>ldu1</i>	INTEGER. The leading dimension of the array u_1 . If <i>jobu1</i> = 'Y', $ldu1 \geq \max(1, p)$.

<i>ldu2</i>	INTEGER. The leading dimension of the array u_2 . If $jobu2 = 'Y'$, $ldu2 \geq \max(1, m-p)$.
<i>ldv1t</i>	INTEGER. The leading dimension of the array $v1t$. If $jobv1t = 'Y'$, $ldv1t \geq \max(1, q)$.
<i>ldv2t</i>	INTEGER. The leading dimension of the array $v2t$. If $jobv2t = 'Y'$, $ldv2t \geq \max(1, m-q)$.
<i>work</i>	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd DOUBLE COMPLEX for zuncsd Workspace array, size $(\max(1, lwork))$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.
<i>rwork</i>	REAL for cuncsd DOUBLE PRECISION for zuncsd Workspace array, size $(\max(1, lrwork))$.
<i>lrwork</i>	INTEGER. The size of the <i>rwork</i> array. Constraints: If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>rwork</i> array, returns this value as the first entry of the <i>rwork</i> array, and no error message related to <i>lrwork</i> is issued by xerbla.
<i>iwork</i>	INTEGER. Workspace array, dimension m .

Output Parameters

<i>theta</i>	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd DOUBLE COMPLEX for zuncsd Array, size (r) , in which $r = \min(p, m-p, q, m-q)$. $C = \text{diag}(\cos(\theta(1)), \dots, \cos(\theta(r)))$, and $S = \text{diag}(\sin(\theta(1)), \dots, \sin(\theta(r)))$.
<i>u1</i>	REAL for sorcsd DOUBLE PRECISION for dorcsd COMPLEX for cuncsd

DOUBLE COMPLEX for zuncsd
Array, size (p).
 If $jobu1 = 'Y'$, $u1$ contains the p -by- p orthogonal/unitary matrix u_1 .

$u2$ REAL for sorcsd
 DOUBLE PRECISION for dorcsd
 COMPLEX for cuncsd
 DOUBLE COMPLEX for zuncsd
Array, size ($ldu2, m-p$).
 If $jobu2 = 'Y'$, $u2$ contains the $(m-p)$ -by- $(m-p)$ orthogonal/unitary matrix u_2 .

$v1t$ REAL for sorcsd
 DOUBLE PRECISION for dorcsd
 COMPLEX for cuncsd
 DOUBLE COMPLEX for zuncsd
Array, size ($ldv1t, *$).
 If $jobv1t = 'Y'$, $v1t$ contains the q -by- q orthogonal matrix v_1^T or unitary matrix v_1^H .

$v2t$ REAL for sorcsd
 DOUBLE PRECISION for dorcsd
 COMPLEX for cuncsd
 DOUBLE COMPLEX for zuncsd
Array, size ($ldv2t, m-q$).
 If $jobv2t = 'Y'$, $v2t$ contains the $(m-q)$ -by- $(m-q)$ orthogonal matrix v_2^T or unitary matrix v_2^H .

$work$ On exit,
 If $info = 0$, $work(1)$ returns the optimal $lwork$.
 If $info > 0$, For ?orcsd, $work(2:r)$ contains the values $phi(1), \dots, phi(r-1)$ that, together with $theta(1), \dots, theta(r)$ define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. $info$ specifies the number of nonzero phi 's.

$rwork$ On exit,
 If $info = 0$, $rwork(1)$ returns the optimal $lrwork$.
 If $info > 0$, For ?uncsd, $rwork(2:r)$ contains the values $phi(1), \dots, phi(r-1)$ that, together with $theta(1), \dots, theta(r)$ define the matrix in

intermediate bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument has an illegal value
 > 0: ?orcsd/?uncsd did not converge. See the description of *work* above for details.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine ?orcsd/?uncsd interface are as follows:

<i>x11</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>q</i>).
<i>x12</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>m-q</i>).
<i>x21</i>	Holds the block of matrix <i>X</i> of size (<i>m-p</i> , <i>q</i>).
<i>x22</i>	Holds the block of matrix <i>X</i> of size (<i>m-p</i> , <i>m-q</i>).
<i>theta</i>	Holds the vector of length $r = \min(p, m-p, q, m-q)$.
<i>u1</i>	Holds the matrix of size (<i>p</i> , <i>p</i>).
<i>u2</i>	Holds the matrix of size (<i>m-p</i> , <i>m-p</i>).
<i>v1t</i>	Holds the matrix of size (<i>q</i> , <i>q</i>).
<i>v2t</i>	Holds the matrix of size (<i>m-q</i> , <i>m-q</i>).
<i>jobsu1</i>	Indicates whether <i>u</i> ₁ is computed. Must be 'Y' or 'O'.
<i>jobsu2</i>	Indicates whether <i>u</i> ₂ is computed. Must be 'Y' or 'O'.
<i>jobv1t</i>	Indicates whether <i>v</i> ₁ ^t is computed. Must be 'Y' or 'O'.
<i>jobv2t</i>	Indicates whether <i>v</i> ₂ ^t is computed. Must be 'Y' or 'O'.
<i>trans</i>	Must be 'N' or 'T'.
<i>signs</i>	Must be 'O' or 'D'.

See Also

[?bbcsd](#)
[xerbla](#)

[?orcsd2by1/?uncsd2by1](#)

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
call sorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
```

```
call dorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
```

```
call cuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
```

```
call zuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
```

```
call orcsd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
```

```
call uncscsd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines `?orcsd2by1/?uncscsd2by1` compute the CS decomposition of an m -by- q matrix X with orthonormal columns that has been partitioned into a 2-by-1 block structure:

$$X = \begin{bmatrix} X_{11} \\ X_{21} \end{bmatrix} = \begin{bmatrix} U_1 & | \\ \hline & U_2 \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & C & 0 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & I \end{bmatrix} V_1^H$$

X_{11} is p -by- q . The orthogonal/unitary matrices u_1 , u_2 , v_1 , and v_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

<i>jobu1</i>	CHARACTER. If equal to 'Y', then u_1 is computed. Otherwise, u_1 is not computed.
<i>jobu2</i>	CHARACTER. If equal to 'Y', then u_2 is computed. Otherwise, u_2 is not computed.
<i>jobv1t</i>	CHARACTER. If equal to 'Y', then v_1^t is computed. Otherwise, v_1^t is not computed.
<i>m</i>	INTEGER. The number of rows and columns of the matrix X .
<i>p</i>	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.
<i>q</i>	INTEGER. The number of columns in x_{11} . $0 \leq q \leq m$.
<i>x11</i>	REAL for <code>sorcsd2by1</code>

	DOUBLE PRECISION for dorcsd2by1 COMPLEX for cuncsd2by1 DOUBLE COMPLEX for zuncsd2by1 Array, size ($ldx11, q$). On entry, the part of the orthogonal matrix whose CSD is desired.
<i>ldx11</i>	INTEGER. The leading dimension of the array <i>x11</i> . $ldx11 \geq \max(1, p)$.
<i>x21</i>	REAL for sorcsd2by1 DOUBLE PRECISION for dorcsd2by1 COMPLEX for cuncsd2by1 DOUBLE COMPLEX for zuncsd2by1 Array, size ($ldx21, q$). On entry, the part of the orthogonal matrix whose CSD is desired.
<i>ldx21</i>	INTEGER. The leading dimension of the array <i>X</i> . $ldx21 \geq \max(1, m - p)$.
<i>ldu1</i>	INTEGER. The leading dimension of the array u_1 . If <i>jobu1</i> = 'Y', $ldu1 \geq \max(1, p)$.
<i>ldu2</i>	INTEGER. The leading dimension of the array u_2 . If <i>jobu2</i> = 'Y', $ldu2 \geq \max(1, m-p)$.
<i>ldv1t</i>	INTEGER. The leading dimension of the array $v1t$. If <i>jobv1t</i> = 'Y', $ldv1t \geq \max(1, q)$.
<i>work</i>	REAL for sorcsd2by1 DOUBLE PRECISION for dorcsd2by1 COMPLEX for cuncsd2by1 DOUBLE COMPLEX for zuncsd2by1 Workspace array, size ($\max(1, lwork)$).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> .
<i>rwork</i>	REAL for cuncsd2by1 DOUBLE PRECISION for zuncsd2by1 Workspace array, size ($\max(1, lrwork)$).
<i>lrwork</i>	INTEGER. The size of the <i>rwork</i> array. Constraints:

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $rwork$ array, returns this value as the first entry of the $rwork$ array, and no error message related to $lwork$ is issued by `xerbla`.

iwork INTEGER. Workspace array, dimension $m - \min(p, m - p, q, m - q)$.

Output Parameters

theta REAL for `sorcsd2by1`
 DOUBLE PRECISION for `dorcsd2by1`
 COMPLEX for `cuncsd2by1`
 DOUBLE COMPLEX for `zuncsd2by1`
 Array, size (r), in which $r = \min(p, m-p, q, m-q)$.
 $C = \text{diag}(\cos(\theta(1)), \dots, \cos(\theta(r)))$, and
 $S = \text{diag}(\sin(\theta(1)), \dots, \sin(\theta(r)))$.

u1 REAL for `sorcsd2by1`
 DOUBLE PRECISION for `dorcsd2by1`
 COMPLEX for `cuncsd2by1`
 DOUBLE COMPLEX for `zuncsd2by1`
 Array, size ($ldu1, p$).
 If $jobu1 = 'Y'$, $u1$ contains the p -by- p orthogonal/unitary matrix u_1 .

u2 REAL for `sorcsd2by1`
 DOUBLE PRECISION for `dorcsd2by1`
 COMPLEX for `cuncsd2by1`
 DOUBLE COMPLEX for `zuncsd2by1`
 Array, size ($ldu2, m-p$).
 If $jobu2 = 'Y'$, $u2$ contains the $(m-p)$ -by- $(m-p)$ orthogonal/unitary matrix u_2 .

v1t REAL for `sorcsd2by1`
 DOUBLE PRECISION for `dorcsd2by1`
 COMPLEX for `cuncsd2by1`
 DOUBLE COMPLEX for `zuncsd2by1`
 Array, size ($ldv1t, q$).
 If $jobv1t = 'Y'$, $v1t$ contains the q -by- q orthogonal matrix v_1^T or unitary matrix v_1^H .

work On exit,
 If $info = 0$, $work(1)$ returns the optimal $lwork$.

If $info > 0$, $work(2:r)$ contains the values $phi(1), \dots, phi(r-1)$ that, together with $theta(1), \dots, theta(r)$ define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. $info$ specifies the number of nonzero phi 's.

$rwork$

On exit,

If $info = 0$, $rwork(1)$ returns the optimal $lrwork$.

If $info > 0$, $rwork(2:r)$ contains the values $phi(1), \dots, phi(r-1)$ that, together with $theta(1), \dots, theta(r)$ define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. $info$ specifies the number of nonzero phi 's.

$info$

INTEGER.

= 0: successful exit

< 0: if $info = -i$, the i -th argument has an illegal value

> 0: $?orcsd2by1/?uncsd2by1$ did not converge. See the description of $work$ above for details.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine $?orcsd2by1/?orcsd2by1$ interface are as follows:

$x11$	Holds the block of matrix X_{11} of size (p, q) .
$x21$	Holds the block of matrix X_{21} of size $(m-p, q)$.
$theta$	Holds the vector of length $r = \min(p, m-p, q, m-q)$.
$u1$	Holds the matrix u_1 of size (p, p) .
$u2$	Holds the matrix u_2 of size $(m-p, m-p)$.
$v1t$	Holds the matrix v_1^T or v_1^H of size (q, q) .
$jobu1$	Indicates whether u_1 is computed. Must be 'Y' or 'O'.
$jobu2$	Indicates whether u_2 is computed. Must be 'Y' or 'O'.
$jobv1t$	Indicates whether v_1^t is computed. Must be 'Y' or 'O'.

See Also

[?bbcsd](#)
[xerbla](#)

Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Symmetric Definite Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
sygv/hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
sygvd/hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
sygvx/hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
spgv/hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
spgvd/hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
spgvx/hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
sbgv/hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.
sbgvd/hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
sbgvx/hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call dsygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call sygv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- `mk1.fi, lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

<i>itype</i>	<p>INTEGER. Must be 1 or 2 or 3.</p> <p>Specifies the problem type to be solved:</p> <p>if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$;</p> <p>if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$;</p> <p>if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.</p>
<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B;</p> <p>If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>a, b, work</i>	<p>REAL for <code>ssygv</code></p> <p>DOUBLE PRECISION for <code>dsygv</code>.</p> <p>Arrays:</p> <p><i>a</i>(<i>lda</i>,*) contains the upper or lower triangle of the symmetric matrix A, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p><i>b</i>(<i>ldb</i>,*) contains the upper or lower triangle of the symmetric positive definite matrix B, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>b</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of <i>a</i>; at least $\max(1, n)$.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of <i>b</i>; at least $\max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>;</p> <p>$lwork \geq \max(1, 3n-1)$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	<p>On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>a</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows:</p> <p>if <i>itype</i> = 1 or 2, $Z^T * B * Z = I$;</p> <p>if <i>itype</i> = 3, $Z^T * \text{inv}(B) * Z = I$;</p> <p>If <i>jobz</i> = 'N', then on exit the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i>, including the diagonal, is destroyed.</p>
<i>b</i>	<p>On exit, if $\text{info} \leq n$, the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.</p>
<i>w</i>	<p>REAL for <i>ssygv</i></p> <p>DOUBLE PRECISION for <i>dsygv</i>.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p>
<i>work(1)</i>	<p>On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value.</p> <p>If <i>info</i> > 0, <i>spotrf/dpotrf</i> or <i>ssyev/dsyev</i> returned an error code:</p> <p>If $\text{info} = i \leq n$, <i>ssyev/dsyev</i> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;</p> <p>If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *sygv* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length <i>n</i> .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use $lwork \geq (nb+2)*n$, where nb is the blocksize for `ssytrd/dsytrd` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of $lwork$ (or $liwork$) for the first run or set $lwork = -1$ ($liwork = -1$).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $work$ ($liwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
call chegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call zhegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call hegv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ;

If `uplo = 'L'`, arrays `a` and `b` store the lower triangles of `A` and `B`.

`n` INTEGER. The order of the matrices `A` and `B` ($n \geq 0$).

`a, b, work` COMPLEX for `chegv`
DOUBLE COMPLEX for `zhegv`.

Arrays:

`a(lda,*)` contains the upper or lower triangle of the Hermitian matrix `A`, as specified by `uplo`.

The second dimension of `a` must be at least $\max(1, n)$.

`b(l db,*)` contains the upper or lower triangle of the Hermitian positive definite matrix `B`, as specified by `uplo`.

The second dimension of `b` must be at least $\max(1, n)$.

`work` is a workspace array, its dimension $\max(1, lwork)$.

`lda` INTEGER. The leading dimension of `a`; at least $\max(1, n)$.

`ldb` INTEGER. The leading dimension of `b`; at least $\max(1, n)$.

`lwork` INTEGER.

The dimension of the array `work`; $lwork \geq \max(1, 2n-1)$.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by [xerbla](#).

See *Application Notes* for the suggested value of `lwork`.

`rwork` REAL for `chegv`
DOUBLE PRECISION for `zhegv`.

Workspace array, size at least $\max(1, 3n-2)$.

Output Parameters

`a` On exit, if `jobz = 'V'`, then if `info = 0`, `a` contains the matrix `Z` of eigenvectors. The eigenvectors are normalized as follows:

if `itype = 1` or `2`, $Z^H * B * Z = I$;

if `itype = 3`, $Z^H * \text{inv}(B) * Z = I$;

If `jobz = 'N'`, then on exit the upper triangle (if `uplo = 'U'`) or the lower triangle (if `uplo = 'L'`) of `A`, including the diagonal, is destroyed.

`b` On exit, if `info ≤ n`, the part of `b` containing the matrix is overwritten by the triangular factor `U` or `L` from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

`w` REAL for `chegv`
DOUBLE PRECISION for `zhegv`.

Array, size at least $\max(1, n)$.

	If $info = 0$, contains the eigenvalues in ascending order.
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th argument has an illegal value. If $info > 0$, $cpotrf/zpotrf$ or $cheev/zheev$ return an error code: If $info = i \leq n$, $cheev/zheev$ fails to converge, and i off-diagonal elements of an intermediate tridiagonal do not converge to zero; If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B can not be completed and no eigenvalues or eigenvectors are computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hegv` interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
w	Holds the vector of length n .
$itype$	Must be 1, 2, or 3. The default value is 1.
$jobz$	Must be 'N' or 'V'. The default value is 'N'.
$uplo$	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where nb is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Syntax

```
call ssygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
call dsygvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info)
call sygvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b, work</i>	REAL for ssygvd DOUBLE PRECISION for dsygvd. Arrays: <i>a</i> (<i>lda</i> ,*) contains the upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

lwork INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1, lwork \geq 1$;

If $jobz = 'N'$ and $n > 1, lwork < 2n + 1$;

If $jobz = 'V'$ and $n > 1, lwork < 2n^2 + 6n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.

Workspace array, its dimension $\max(1, lwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1, liwork \geq 1$;

If $jobz = 'N'$ and $n > 1, liwork \geq 1$;

If $jobz = 'V'$ and $n > 1, liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a On exit, if $jobz = 'V'$, then if $info = 0$, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or $2, Z^T * B * Z = I$;

if $itype = 3, Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of *A*, including the diagonal, is destroyed.

b On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

w REAL for `ssygvd`

DOUBLE PRECISION for `dsygv`.

Array, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

work(1) On exit, if $info = 0$, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if $info = 0$, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, an error code is returned as specified below.

- For $info \leq n$:
 - If $info = i$ and $jobz = 'N'$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
 - If $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.
- For $info > n$:
 - If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sygv` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set $lwork = -1$ ($liwork = -1$).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem using a divide and conquer method.

Syntax

```
call chegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, lrwork,
iwork, liwork, info)
```

```
call zhegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, lrwork,
iwork, liwork, info)
```

```
call hegvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here *A* and *B* are assumed to be Hermitian and *B* is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

a, *b*, *work* COMPLEX for `chegvd`
DOUBLE COMPLEX for `zhegvd`.

Arrays:
a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix *A*, as specified by *uplo*.
The second dimension of *a* must be at least $\max(1, n)$.
b(*ldb*,*) contains the upper or lower triangle of the Hermitian positive definite matrix *B*, as specified by *uplo*.
The second dimension of *b* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

lwork INTEGER.
The dimension of the array *work*.
Constraints:
If $n \leq 1$, $lwork \geq 1$;
If *jobz* = 'N' and $n > 1$, $lwork \geq n + 1$;
If *jobz* = 'V' and $n > 1$, $lwork \geq n^2 + 2n$.
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for `chegvd`
DOUBLE PRECISION for `zhegvd`.
Workspace array, size $\max(1, lrwork)$.

lrwork INTEGER.
The dimension of the array *rwork*.
Constraints:
If $n \leq 1$, $lrwork \geq 1$;
If *jobz* = 'N' and $n > 1$, $lrwork \geq n$;
If *jobz* = 'V' and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.
If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.
Workspace array, size $\max(1, liwork)$.

liwork INTEGER.
The dimension of the array *iwork*.
Constraints:
If $n \leq 1, liwork \geq 1$;
If $jobz = 'N'$ and $n > 1, liwork \geq 1$;
If $jobz = 'V'$ and $n > 1, liwork \geq 5n+3$.
If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a On exit, if $jobz = 'V'$, then if $info = 0$, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:
if $itype = 1$ or $2, Z^H * B * Z = I$;
if $itype = 3, Z^H * \text{inv}(B) * Z = I$;
If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of *A*, including the diagonal, is destroyed.

b On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

w REAL for `chegvd`
DOUBLE PRECISION for `zhegvd`.
Array, size at least $\max(1, n)$.
If $info = 0$, contains the eigenvalues in ascending order.

work(1) On exit, if $info = 0$, then *work*(1) returns the required minimal size of *lwork*.

rwork(1) On exit, if $info = 0$, then *rwork*(1) returns the required minimal size of *lrwork*.

iwork(1) On exit, if $info = 0$, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the *i*-th argument had an illegal value.

If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $mod(info, n+1)$.

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hegvd` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set $lwork = -1$ ($liwork = -1, lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value ($work(1), iwork(1), rwork(1)$) for subsequent runs.

If you set $lwork = -1$ ($liwork = -1, lrwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, iwork, ifail, info)
```

```
call sygvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	<p>INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:</p> <p>if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$;</p> <p>if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$;</p> <p>if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.</p>
<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval:</p> $vl < \lambda(i) \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B;</p> <p>If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>a, b, work</i>	<p>REAL for ssygvx</p> <p>DOUBLE PRECISION for dsygvx.</p> <p>Arrays:</p> <p><i>a</i>(<i>lda</i>,*) contains the upper or lower triangle of the symmetric matrix A, as specified by <i>uplo</i>.</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>

$b(ldb,*)$ contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by $uplo$.

The second dimension of b must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
vl, vu	REAL for <code>ssygvx</code> DOUBLE PRECISION for <code>dsygvx</code> . If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If $range = 'A'$ or $'I'$, vl and vu are not referenced.
il, iu	INTEGER. If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If $range = 'A'$ or $'V'$, il and iu are not referenced.
$abstol$	REAL for <code>ssygvx</code> DOUBLE PRECISION for <code>dsygvx</code> . The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
ldz	INTEGER. The leading dimension of the output array z . Constraints: $ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.
$lwork$	INTEGER. The dimension of the array $work$; $lwork < \max(1, 8n)$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla . See <i>Application Notes</i> for the suggested value of $lwork$.
$iwork$	INTEGER. Workspace array, size at least $\max(1, 5n)$.

Output Parameters

a	On exit, the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of A , including the diagonal, is overwritten.
-----	--

<i>b</i>	<p>On exit, if $info \leq n$, the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.</p>
<i>m</i>	<p>INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.</p>
<i>w, z</i>	<p>REAL for ssygvx DOUBLE PRECISION for dsygvx.</p> <p>Arrays: <i>w</i>(*), size at least $\max(1, n)$. The first <i>m</i> elements of <i>w</i> contain the selected eigenvalues in ascending order. <i>z</i>(<i>ldz</i>,*) . The second dimension of <i>z</i> must be at least $\max(1, m)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>). The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^T * B * Z = I$; if <i>itype</i> = 3, $Z^T * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.</p>
<i>work</i> (1)	<p>On exit, if <i>info</i> = 0, then <i>work</i>(1) returns the required minimal size of <i>lwork</i>.</p>
<i>ifail</i>	<p>INTEGER. Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.</p>
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = -<i>i</i>, the <i>i</i>th argument had an illegal value. If <i>info</i> > 0, spotrf/dpotrf and sseyvx/dsyevx returned an error code:</p>

If $info = i \leq n$, $ssyevx/dsyevx$ failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *sygvx* interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>z</i>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

	<p>if $itype = 1$, the problem type is $A*x = \lambda*B*x$;</p> <p>if $itype = 2$, the problem type is $A*B*x = \lambda*x$;</p> <p>if $itype = 3$, the problem type is $B*A*x = \lambda*x$.</p>
<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If $jobz = 'N'$, then compute eigenvalues only.</p> <p>If $jobz = 'V'$, then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If $range = 'A'$, the routine computes all eigenvalues.</p> <p>If $range = 'V'$, the routine computes eigenvalues $lambda(i)$ in the half-open interval:</p> <p>$vl < lambda(i) \leq vu$.</p> <p>If $range = 'I'$, the routine computes eigenvalues with indices il to iu.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If $uplo = 'U'$, arrays a and b store the upper triangles of A and B;</p> <p>If $uplo = 'L'$, arrays a and b store the lower triangles of A and B.</p>
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b, work</i>	<p>COMPLEX for <code>chegvx</code></p> <p>DOUBLE COMPLEX for <code>zhegvx</code>.</p> <p>Arrays:</p> <p>$a(lda,*)$ contains the upper or lower triangle of the Hermitian matrix A, as specified by $uplo$.</p> <p>The second dimension of a must be at least $\max(1, n)$.</p> <p>$b(l db,*)$ contains the upper or lower triangle of the Hermitian positive definite matrix B, as specified by $uplo$.</p> <p>The second dimension of b must be at least $\max(1, n)$.</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; at least $\max(1, n)$.
<i>vl, vu</i>	<p>REAL for <code>chegvx</code></p> <p>DOUBLE PRECISION for <code>zhegvx</code>.</p> <p>If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If $range = 'A'$ or $'I'$, vl and vu are not referenced.</p>
<i>il, iu</i>	INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$

if $n = 0$.

If $range = 'A'$ or $'V'$, il and iu are not referenced.

abstol

REAL for chegvx

DOUBLE PRECISION for zhegvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z . Constraints:

$ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*; $lwork \geq \max(1, 2n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

rwork

REAL for chegvx

DOUBLE PRECISION for zhegvx.

Workspace array, size at least $\max(1, 7n)$.

iwork

INTEGER.

Workspace array, size at least $\max(1, 5n)$.

Output Parameters

a

On exit, the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of A , including the diagonal, is overwritten.

b

On exit, if $info \leq n$, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$,

$m = iu - il + 1$.

w

REAL for chegvx

DOUBLE PRECISION for zhegvx.

Array, size at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues in ascending order.

 z

COMPLEX for chegvx

DOUBLE COMPLEX for zhegvx.

Array $z(ldz,*)$. The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^H * B * Z = I$;

if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

 $work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

 $ifail$

INTEGER.

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

 $info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th argument had an illegal value.

If $info > 0$, cpotrf/zpotrf and cheevx/zheevx returned an error code:

If $info = i \leq n$, cheevx/zheevx failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hegvx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n, n) .
<i>w</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector of length <i>n</i> .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *C* to tridiagonal form, where *C* is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{lamch}('S')$.

For optimum performance use $lwork \geq (nb+1) * n$, where *nb* is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call dspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call spgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x.$$

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp, work</i>	REAL for sspgv DOUBLE PRECISION for dspgv.

Arrays:

ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.

work(*) is a workspace array, size at least $\max(1, 3n)$.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

Output Parameters

ap On exit, the contents of *ap* are overwritten.

bp On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as *B*.

w, z REAL for *sspgv*
DOUBLE PRECISION for *dspgv*.

Arrays:

w(*), size at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T * B * Z = I$;

if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;

If *jobz* = 'N', then *z* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, *ssptrfs/dpptrfs* and *sspev/dspev* returned an error code:

If *info* = $i \leq n$, *sspev/dspev* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = $n + i$, for $1 \leq i \leq n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `spgv` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call zhpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call hpgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

`itype` INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:
 if `itype = 1`, the problem type is $A^*x = \lambda B^*x$;
 if `itype = 2`, the problem type is $A^*B^*x = \lambda^*x$;
 if `itype = 3`, the problem type is $B^*A^*x = \lambda^*x$.

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i>;</p> <p>If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i>.</p>
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp, work</i>	<p>COMPLEX for <i>chpgv</i></p> <p>DOUBLE COMPLEX for <i>zhpgv</i>.</p> <p>Arrays:</p> <p><i>ap</i>(*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i>(*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>work</i>(*) is a workspace array, size at least $\max(1, 2n-1)$.</p>
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>rwork</i>	<p>REAL for <i>chpgv</i></p> <p>DOUBLE PRECISION for <i>zhpgv</i>.</p> <p>Workspace array, size at least $\max(1, 3n-2)$.</p>

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>w</i>	<p>REAL for <i>chpgv</i></p> <p>DOUBLE PRECISION for <i>zhpgv</i>.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p>
<i>z</i>	<p>COMPLEX for <i>chpgv</i></p> <p>DOUBLE COMPLEX for <i>zhpgv</i>.</p> <p>Array <i>z</i>(<i>ldz</i>,*).</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows:</p>

if *itype* = 1 or 2, $Z^H * B * Z = I$;
 if *itype* = 3, $Z^H * \text{inv}(B) * Z = I$;
 If *jobz* = 'N', then *z* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*-th argument had an illegal value.If *info* > 0, *cpptrf/zpptrf* and *chpev/zhpev* returned an error code:If *info* = *i* ≤ *n*, *chpev/zhpev* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hpgv* interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>bp</i>	Holds the array <i>B</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
call sspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call dspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call spgvd(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	<p>INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:</p> <p>if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$;</p> <p>if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$;</p> <p>if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.</p>
<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B;</p> <p>If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>ap, bp, work</i>	<p>REAL for <code>sspgvd</code></p> <p>DOUBLE PRECISION for <code>dspgvd</code>.</p> <p>Arrays:</p> <p><i>ap</i>(*) contains the packed upper or lower triangle of the symmetric matrix A, as specified by <i>uplo</i>.</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i>(*) contains the packed upper or lower triangle of the symmetric matrix B, as specified by <i>uplo</i>.</p> <p>The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>Constraints:</p> <p>If $n \leq 1$, $lwork \geq 1$;</p>

If $jobz = 'N'$ and $n > 1$, $lwork \geq 2n$;

If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n^2 + 6n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, dimension $\max(1, lwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;

If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as *B*.

w, z

REAL for *sspgv*

DOUBLE PRECISION for *dspgv*.

Arrays:

w(*), size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T * B * Z = I$;

if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then *z* is not referenced.

work(1)

On exit, if $info = 0$, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0, *sppturf/dppturf* and *sspevd/dspevd* returned an error code:

If *info* = *i* ≤ *n*, *sspevd/dspevd* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *spgv*d interface are the following:

ap Holds the array *A* of size $(n*(n+1)/2)$.

bp Holds the array *B* of size $(n*(n+1)/2)$.

w Holds the vector with the number of elements *n*.

z Holds the matrix *Z* of size (n, n) .

itype Must be 1, 2, or 3. The default value is 1.

uplo Must be 'U' or 'L'. The default value is 'U'.

jobz Restored based on the presence of the argument *z* as follows:
jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
call chpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork, info)
```

```
call zhpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork,
            iwork, liwork, info)
```

```
call hpgvd(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ap, bp, work</i>	COMPLEX for chpgvd DOUBLE COMPLEX for zhpgvd. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.

bp(*) contains the packed upper or lower triangle of the Hermitian matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.

work is a workspace array, its dimension $\max(1, lwork)$.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

lwork INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1, lwork \geq 1$;

If *jobz* = 'N' and $n > 1, lwork \geq n$;

If *jobz* = 'V' and $n > 1, lwork \geq 2n$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

rwork REAL for *chpgvd*

DOUBLE PRECISION for *zhpgvd*.

Workspace array, its dimension $\max(1, lrwork)$.

lrwork INTEGER.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1, lrwork \geq 1$;

If *jobz* = 'N' and $n > 1, lrwork \geq n$;

If *jobz* = 'V' and $n > 1, lrwork \geq 2n^2 + 5n + 1$.

If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1, liwork \geq 1$;

If *jobz* = 'N' and $n > 1, liwork \geq 1$;

If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>w</i>	REAL for <code>chpgvd</code> DOUBLE PRECISION for <code>zhpgvd</code> . Array, size at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for <code>chpgvd</code> DOUBLE COMPLEX for <code>zhpgvd</code> . Array $z(ldz, *)$. The second dimension of <i>z</i> must be at least $\max(1, n)$. If $jobz = 'V'$, then if $info = 0$, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows: if $itype = 1$ or 2 , $Z^H * B * Z = I$; if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$; If $jobz = 'N'$, then <i>z</i> is not referenced.
<i>work</i> (1)	On exit, if $info = 0$, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>rwork</i> (1)	On exit, if $info = 0$, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if $info = 0$, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th argument had an illegal value. If $info > 0$, <code>cpptrf/zpptrf</code> and <code>chpevd/zhpevd</code> returned an error code: If $info = i \leq n$, <code>chpevd/zhpevd</code> failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpgvd` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of `lwork` (`liwork` or `lrwork`) for the first run or set `lwork = -1` (`liwork = -1, lrwork = -1`).

If you choose the first option and set any of admissible `lwork` (`liwork` or `lrwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`) on exit. Use this value (`work(1)`, `iwork(1)`, `rwork(1)`) for subsequent runs.

If you set `lwork = -1` (`liwork = -1, lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
call sspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info)
```

```
call dspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info)
```

```
call spgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda^*(i)$ in the half-open interval: $vl < \lambda^*(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ap, bp, work</i>	REAL for <code>sspgvx</code> DOUBLE PRECISION for <code>dspgvx</code> . Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> (*) contains the packed upper or lower triangle of the symmetric matrix B , as specified by <i>uplo</i> . The size of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.

work(*) is a workspace array, size at least $\max(1, 8n)$.

vl, vu

REAL for *sspgvx*

DOUBLE PRECISION for *dspgvx*.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$

if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for *sspgvx*

DOUBLE PRECISION for *dspgvx*.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array *z*. Constraints:

$ldz \geq 1$; if *jobz* = 'V', $ldz \geq \max(1, n)$.

iwork

INTEGER.

Workspace array, size at least $\max(1, 5n)$.

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as *B*.

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',

$m = iu - il + 1$.

w, z

REAL for *sspgvx*

DOUBLE PRECISION for *dspgvx*.

Arrays:

w(*), size at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*) .

The second dimension of *z* must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T * B * Z = I$;

if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ifail

INTEGER.

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, *spptrf*/*dpptf* and *sspevx*/*dspevx* returned an error code:

If $info = i \leq n$, *sspevx*/*dspevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *spgvx* interface are the following:

<i>ap</i>	Holds the array A of size $(n*(n+1)/2)$.
<i>bp</i>	Holds the array B of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<i>ifail</i>	Holds the vector with the number of elements n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.

<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ is used instead, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to $2 * \text{lamch}('S')$.

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
call chpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, rwork, iwork, ifail, info)

call zhpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, rwork, iwork, ifail, info)

call hpgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>itype</i>	<p>INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:</p> <p>if <i>itype</i> = 1, the problem type is $A^*x = \lambda B^*x$;</p> <p>if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$;</p> <p>if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.</p>
<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval:</p> $vl < \lambda(i) \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B;</p> <p>If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B.</p>
<i>n</i>	<p>INTEGER. The order of the matrices A and B ($n \geq 0$).</p>
<i>ap, bp, work</i>	<p>COMPLEX for <code>chpgvx</code></p> <p>DOUBLE COMPLEX for <code>zhpgvx</code>.</p> <p>Arrays:</p> <p><i>ap</i>(*) contains the packed upper or lower triangle of the Hermitian matrix A, as specified by <i>uplo</i>.</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i>(*) contains the packed upper or lower triangle of the Hermitian matrix B, as specified by <i>uplo</i>.</p> <p>The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.</p>

	<i>work</i> (*) is a workspace array, size at least $\max(1, 2n)$.
<i>vl, vu</i>	<p>REAL for <code>chpgvx</code></p> <p>DOUBLE PRECISION for <code>zhpgvx</code>.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>REAL for <code>chpgvx</code></p> <p>DOUBLE PRECISION for <code>zhpgvx</code>.</p> <p>The absolute error tolerance for the eigenvalues.</p> <p>See <i>Application Notes</i> for more information.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.</p>
<i>rwork</i>	<p>REAL for <code>chpgvx</code></p> <p>DOUBLE PRECISION for <code>zhpgvx</code>.</p> <p>Workspace array, size at least $\max(1, 7n)$.</p>
<i>iwork</i>	<p>INTEGER.</p> <p>Workspace array, size at least $\max(1, 5n)$.</p>

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>m</i>	<p>INTEGER. The total number of eigenvalues found,</p> <p>$0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I',</p> <p>$m = iu - il + 1$.</p>
<i>w</i>	<p>REAL for <code>chpgvx</code></p> <p>DOUBLE PRECISION for <code>zhpgvx</code>.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p>

<i>z</i>	<p>COMPLEX for <code>chpgvx</code></p> <p>DOUBLE COMPLEX for <code>zhpgvx</code>.</p> <p>Array <i>z</i>(<i>ldz</i>,*).</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>). The eigenvectors are normalized as follows:</p> <p>if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$;</p> <p>if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$;</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p> <p>If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p> <p>Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i>; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.</p>
<i>ifail</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge.</p> <p>If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value.</p> <p>If <i>info</i> > 0, <code>cpptrf/zpptrf</code> and <code>chpevx/zhpevx</code> returned an error code:</p> <p>If <i>info</i> = <i>i</i> ≤ <i>n</i>, <code>chpevx/zhpevx</code> failed to converge, and <i>i</i> eigenvectors failed to converge. Their indices are stored in the array <i>ifail</i>;</p> <p>If <i>info</i> = <i>n</i> + <i>i</i>, for $1 \leq i \leq n$, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hpgvx` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>bp</i>	Holds the array <i>B</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .

<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) , where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector with the number of elements <i>n</i> .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: $jobz = 'V'$, if <i>z</i> is present, $jobz = 'N'$, if <i>z</i> is omitted. Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 \cdot \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 \cdot \text{?lamch}('S')$.

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call dsbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call sbgv(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb, work</i>	REAL for <code>ssbgv</code> DOUBLE PRECISION for <code>dsbgv</code> Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, <i>dimension</i> at least $\max(1, 3n)$
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

<i>bb</i>	On exit, contains the factor S from the split Cholesky factorization $B = S^T * S$, as returned by pbstf/pbstf .
<i>w, z</i>	<p>REAL for <code>ssbgv</code></p> <p>DOUBLE PRECISION for <code>dsbgv</code></p> <p>Arrays:</p> <p><i>w</i>(*), size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p> <p><i>z</i>(<i>ldz</i>,*) .</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix Z of eigenvectors, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>). The eigenvectors are normalized so that $Z^T * B * Z = I$.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value.</p> <p>If <i>info</i> > 0, and</p> <p>if $i \leq n$, the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;</p> <p>if $info = n + i$, for $1 \leq i \leq n$, then pbstf/pbstf returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbgv` interface are the following:

<i>ab</i>	Holds the array A of size $(ka+1, n)$.
<i>bb</i>	Holds the array B of size $(kb+1, n)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix Z of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```
call chbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call zhbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call hbgv(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are Hermitian and banded matrices, and matrix B is also positive definite.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb, work</i>	COMPLEX for chbgv DOUBLE COMPLEX for zhbgv Arrays: <i>ab(ldab,*)</i> is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb(ldbb,*)</i> is an array containing either upper or lower triangular part of the Hermitian matrix B (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work(*)</i> is a workspace array, dimension at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.

ldbb INTEGER. The leading dimension of the array *bb*; must be at least $kb+1$.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

rwork REAL for chbgv
DOUBLE PRECISION for zhbqv.
Workspace array, size at least $\max(1, 3n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^H * S$, as returned by [pbstf/pbstf](#).

w REAL for chbgv
DOUBLE PRECISION for zhbqv.
Array, size at least $\max(1, n)$.
If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chbgv
DOUBLE COMPLEX for zhbqv.
Array *z*(*ldz*,*).
The second dimension of *z* must be at least $\max(1, n)$.
If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H * B * Z = I$.
If *jobz* = 'N', then *z* is not referenced.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th argument had an illegal value.
If *info* > 0, and
if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;
if $info = n + i$, for $1 \leq i \leq n$, then [pbstf/pbstf](#) returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *hbqv* interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size $(ka+1,n)$.
<i>bb</i>	Holds the array <i>B</i> of size $(kb+1,n)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call ssgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork,
liwork, info)
```

```
call dsbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork,
liwork, info)
```

```
call sbgvd(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here *A* and *B* are assumed to be symmetric and banded, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i>

($ka \geq 0$).

kb INTEGER. The number of super- or sub-diagonals in *B* ($kb \geq 0$).

ab, *bb*, *work* REAL for *ssbgvd*

DOUBLE PRECISION for *dsbgvd*

Arrays:

ab(*ldab*,*) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least $\max(1, n)$.

bb(*ldb*,*) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

ldab INTEGER. The leading dimension of the array *ab*; must be at least $ka+1$.

ldb INTEGER. The leading dimension of the array *bb*; must be at least $kb+1$.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

lwork INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $lwork \geq 3n$;

If *jobz* = 'V' and $n > 1$, $lwork \geq 2n^2+5n+1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If *jobz* = 'N' and $n > 1$, $liwork \geq 1$;

If *jobz* = 'V' and $n > 1$, $liwork \geq 5n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^T * S$, as returned by pbstf/pbstf .
<i>w, z</i>	<p>REAL for <code>ssbgvd</code></p> <p>DOUBLE PRECISION for <code>dsbgvd</code></p> <p>Arrays:</p> <p><i>w</i>(*), size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p> <p><i>z</i>(<i>ldz</i>,*).</p> <p>The second dimension of <i>z</i> must be at least $\max(1, n)$.</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>). The eigenvectors are normalized so that $Z^T * B * Z = I$.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work</i> (1)	On exit, if <i>info</i> = 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th argument had an illegal value.</p> <p>If <i>info</i> > 0, and</p> <p>if $i \leq n$, the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero;</p> <p>if $info = n + i$, for $1 \leq i \leq n$, then pbstf/pbstf returned $info = i$ and <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbgvd` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size $(ka+1,n)$.
<i>bb</i>	Holds the array <i>B</i> of size $(kb+1,n)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
call chbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call zhbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call hbgvd(ab, bb, w [,uplo] [,z] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here *A* and *B* are assumed to be Hermitian and banded, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>jobz</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i>;</p> <p>If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i>.</p>
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	<p>INTEGER. The number of super- or sub-diagonals in <i>A</i></p> <p>($ka \geq 0$).</p>
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb, work</i>	<p>COMPLEX for <code>chbgvd</code></p> <p>DOUBLE COMPLEX for <code>zhbgvd</code></p> <p>Arrays:</p> <p><i>ab</i>(<i>ldab</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of the array <i>ab</i> must be at least $\max(1, n)$.</p> <p><i>bb</i>(<i>ldbb</i>,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format.</p> <p>The second dimension of the array <i>bb</i> must be at least $\max(1, n)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	<p>INTEGER. The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.</p>
<i>lwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>work</i>.</p> <p>Constraints:</p> <p>If $n \leq 1$, $lwork \geq 1$;</p> <p>If <i>jobz</i> = 'N' and $n > 1$, $lwork \geq n$;</p> <p>If <i>jobz</i> = 'V' and $n > 1$, $lwork \geq 2n^2$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>lrwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.</p>

<i>rwork</i>	<p>REAL for chbgvd</p> <p>DOUBLE PRECISION for zhbgvd.</p> <p>Workspace array, size $\max(1, lrwork)$.</p>
<i>lrwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>rwork</i>.</p> <p>Constraints:</p> <p>If $n \leq 1, lrwork \geq 1$;</p> <p>If $jobz = 'N'$ and $n > 1, lrwork \geq n$;</p> <p>If $jobz = 'V'$ and $n > 1, lrwork \geq 2n^2 + 5n + 1$.</p> <p>If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>lrwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.</p>
<i>iwork</i>	<p>INTEGER.</p> <p>Workspace array, size $\max(1, liwork)$.</p>
<i>liwork</i>	<p>INTEGER.</p> <p>The dimension of the array <i>iwork</i>.</p> <p>Constraints:</p> <p>If $n \leq 1, liwork \geq 1$;</p> <p>If $jobz = 'N'$ and $n > 1, liwork \geq 1$;</p> <p>If $jobz = 'V'$ and $n > 1, liwork \geq 5n + 3$.</p> <p>If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i>, <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>lrwork</i> or <i>liwork</i> is issued by xerbla. See <i>Application Notes</i> for details.</p>

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^H * S$, as returned by pbstf/pbstf .
<i>w</i>	<p>REAL for chbgvd</p> <p>DOUBLE PRECISION for zhbgvd.</p> <p>Array, size at least $\max(1, n)$.</p> <p>If $info = 0$, contains the eigenvalues in ascending order.</p>
<i>z</i>	<p>COMPLEX for chbgvd</p> <p>DOUBLE COMPLEX for zhbgvd</p>

Array $z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H * B * Z = I$.

If $jobz = 'N'$, then z is not referenced.

$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$rwork(1)$	On exit, if $info = 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th argument had an illegal value.</p> <p>If $info > 0$, and</p> <p>if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;</p> <p>if $info = n + i$, for $1 \leq i \leq n$, then <code>pbstf/pbstf</code> returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.</p>

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbgvd` interface are the following:

ab	Holds the array A of size $(ka+1, n)$.
bb	Holds the array B of size $(kb+1, n)$.
w	Holds the vector with the number of elements n .
z	Holds the matrix Z of size (n, n) .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of $lwork$ ($liwork$ or $lrwork$) for the first run or set $lwork = -1$ ($liwork = -1$, $lrwork = -1$).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
call ssbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
  abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call dsbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
  abstol, m, w, z, ldz, work, iwork, ifail, info)
```

```
call sbgvx(ab, bb, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
  [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues in range <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, arrays ab and bb store the upper triangles of A and B ;
 If $uplo = 'L'$, arrays ab and bb store the lower triangles of A and B .

n INTEGER. The order of the matrices A and B ($n \geq 0$).

ka INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).

kb INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).

$ab, bb, work$ REAL for `ssbgvx`
 DOUBLE PRECISION for `dsbgvx`

Arrays:
 $ab(ldab,*)$ is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by $uplo$) in band storage format.
 The second dimension of the array ab must be at least $\max(1, n)$.
 $bb(lddb,*)$ is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by $uplo$) in band storage format.
 The second dimension of the array bb must be at least $\max(1, n)$.
 $work(*)$ is a workspace array, size $(7*n)$.

$ldab$ INTEGER. The leading dimension of the array ab ; must be at least $ka+1$.

$ldbb$ INTEGER. The leading dimension of the array bb ; must be at least $kb+1$.

vl, vu REAL for `ssbgvx`
 DOUBLE PRECISION for `dsbgvx`.
 If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.
 Constraint: $vl < vu$.
 If $range = 'A'$ or $'I'$, vl and vu are not referenced.

il, iu INTEGER.
 If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.
 Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.
 If $range = 'A'$ or $'V'$, il and iu are not referenced.

$abstol$ REAL for `ssbgvx`
 DOUBLE PRECISION for `dsbgvx`.
 The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

ldq INTEGER. The leading dimension of the output array *q*; $ldq < 1$.
 If *jobz* = 'V', $ldq < \max(1, n)$.

iwork INTEGER.
 Workspace array, size $(5*n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^T * S$, as returned by [pbstf/pbstf](#).

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',
 $m = iu - il + 1$.

w, z, q REAL for *ssbgvx*
 DOUBLE PRECISION for *dsbgvx*
Arrays:
w(*), size at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues in ascending order.
z(*ldz*,*) .
 The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T * B * Z = I$.
 If *jobz* = 'N', then *z* is not referenced.
q(*ldq*,*) .
 The second dimension of *q* must be at least $\max(1, n)$.
 If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of $A * x = lambda * B * x$ to standard form, that is, $C * x = lambda * x$ and consequently *C* to tridiagonal form.
 If *jobz* = 'N', then *q* is not referenced.

ifail INTEGER.
 Array, size (*m*).
 If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then `pbstf/pbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `sbgvx` interface are the following:

<code>ab</code>	Holds the array A of size $(ka+1, n)$.
<code>bb</code>	Holds the array B of size $(kb+1, n)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>ifail</code>	Holds the vector with the number of elements n .
<code>q</code>	Holds the matrix Q of size (n, n) .
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>vl</code>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<code>vu</code>	Default value for this element is $vu = \text{HUGE}(vu)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this element is $abstol = 0.0_WP$.
<code>jobz</code>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <code>ifail</code> or <code>q</code> is present and z is omitted.
<code>range</code>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{?lamch}('S')$.

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```
call chbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call zhbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

```
call hbgvx(ab, bb, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

jobz CHARACTER*1. Must be 'N' or 'V'.
 If *jobz* = 'N', then compute eigenvalues only.
 If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.
 If *range* = 'A', the routine computes all eigenvalues.
 If *range* = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval:
 $vl < \lambda(i) \leq vu$.
 If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo CHARACTER*1. Must be 'U' or 'L'.
 If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of A and B ;

If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

n INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ka INTEGER. The number of super- or sub-diagonals in *A* ($ka \geq 0$).

kb INTEGER. The number of super- or sub-diagonals in *B* ($kb \geq 0$).

ab, bb, work COMPLEX for *chbgvx*
DOUBLE COMPLEX for *zhbgvx*

Arrays:

ab(ldab,)* is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format. The second dimension of the array *ab* must be at least $\max(1, n)$.

bb(lddb,)* is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format. The second dimension of the array *bb* must be at least $\max(1, n)$.

work()* is a workspace array, size at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*; must be at least $ka+1$.

ldbb INTEGER. The leading dimension of the array *bb*; must be at least $kb+1$.

vl, vu REAL for *chbgvx*
DOUBLE PRECISION for *zhbgvx*.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for *chbgvx*
DOUBLE PRECISION for *zhbgvx*.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

ldq INTEGER. The leading dimension of the output array *q*; $ldq \geq 1$. If *jobz* = 'V', $ldq \geq \max(1, n)$.

rwork REAL for `chbgvx`
 DOUBLE PRECISION for `zhbgvx`.
 Workspace array, size at least $\max(1, 7n)$.

iwork INTEGER.
 Workspace array, size at least $\max(1, 5n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^H * S$, as returned by `pbstf/pbstf`.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',
 $m = iu - il + 1$.

w REAL for `chbgvx`
 DOUBLE PRECISION for `zhbgvx`.
 Array *w*(*), size at least $\max(1, n)$.
 If *info* = 0, contains the eigenvalues in ascending order.

z, q COMPLEX for `chbgvx`
 DOUBLE COMPLEX for `zhbgvx`
 Arrays:
z(*ldz*,*).
 The second dimension of *z* must be at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^H * B * Z = I$.
 If *jobz* = 'N', then *z* is not referenced.
q(*ldq*,*).
 The second dimension of *q* must be at least $\max(1, n)$.
 If *jobz* = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently *C* to tridiagonal form.
 If *jobz* = 'N', then *q* is not referenced.

ifail INTEGER.
 Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then `pbstf/pbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `hbgvx` interface are the following:

<i>ab</i>	Holds the array A of size $(ka+1, n)$.
<i>bb</i>	Holds the array B of size $(kb+1, n)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix Z of size (n, n) .
<i>ifail</i>	Holds the vector with the number of elements n .
<i>q</i>	Holds the matrix Q of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vu)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_WP$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if $ifail$ or q is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments vl , vu , il , iu as follows:

$range = 'V'$, if one of or both vl and vu are present,

$range = 'I'$, if one of or both il and iu are present,

$range = 'A'$, if none of vl , vu , il , iu is present,

Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * ?lamch('S')$.

Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Nonsymmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
ggev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
ggevxx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

```
call sgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphas, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)
```

```
call dgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphas, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)
```

```
call cgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
```

```
call zgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl, ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
```

```
call gges(a, b, alphas, alpha, beta [,vsl] [,vsr] [,select] [,sdim] [,info])
```

call `gges(a, b, alpha, beta [, vs1] [, vsr] [,select] [,sdim] [,info])`

Include Files

- `mkl.fi`, `lapack.f90`

Description

The `?gges` routine computes the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors ($vs1$ and vsr) for a pair of n -by- n real/complex nonsymmetric matrices (A,B) . This gives the generalized Schur factorization

$$(A,B) = (vs1 * S * vsr^H, vs1 * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T . The leading columns of $vs1$ and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

If only the generalized eigenvalues are needed, use the driver `ggev` instead, which is faster.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $alpha / beta = w$, such that $A - w * B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta=0$ or for both being zero. A pair of matrices (S,T) is in the generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S are "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

The `?gges` routine replaces the deprecated `?gegs` routine.

Input Parameters

<code>jobvs1</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>jobvs1</code> = 'N', then the left Schur vectors are not computed. If <code>jobvs1</code> = 'V', then the left Schur vectors are computed.
<code>jobvsr</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>jobvsr</code> = 'N', then the right Schur vectors are not computed. If <code>jobvsr</code> = 'V', then the right Schur vectors are computed.
<code>sort</code>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. If <code>sort</code> = 'N', then eigenvalues are not ordered. If <code>sort</code> = 'S', eigenvalues are ordered (see <code>selctg</code>).
<code>selctg</code>	LOGICAL FUNCTION of three REAL arguments for real flavors.

LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.

selctg must be declared EXTERNAL in the calling subroutine.

If *sort* = 'S', *selctg* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *selctg* is not referenced.

For real flavors:

An eigenvalue $(\text{alphan}(j) + \text{alphai}(j))/\text{betan}(j)$ is selected if *selctg*(*alphan*(*j*), *alphai*(*j*), *betan*(*j*)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *selctg*(*alphan*(*j*), *alphai*(*j*), *betan*(*j*)) = .TRUE. after ordering. In this case *info* is set to *n*+2 .

For complex flavors:

An eigenvalue $\text{alpha}(j) / \text{beta}(j)$ is selected if *selctg*(*alpha*(*j*), *beta*(*j*)) is true.

Note that a selected complex eigenvalue may no longer satisfy *selctg*(*alpha*(*j*), *beta*(*j*)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

n

INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* ($n \geq 0$).

a, *b*, *work*

REAL for *sgges*

DOUBLE PRECISION for *dsges*

COMPLEX for *cgges*

DOUBLE COMPLEX for *zsges*.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least max(1, *n*).

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of the array *a*. Must be at least max(1, *n*).

ldb

INTEGER. The leading dimension of the array *b*. Must be at least max(1, *n*).

ldvsl, *ldvsr*

INTEGER. The leading dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:

ldvsl ≥ 1 . If *jobvsl* = 'V', *ldvsl* \geq max(1, *n*).

ldvsr ≥ 1 . If *jobvsr* = 'V', *ldvsr* \geq max(1, *n*).

lwork

INTEGER.

The dimension of the array *work*.

$lwork \geq \max(1, 8n+16)$ for real flavors;

$lwork \geq \max(1, 2n)$ for complex flavors.

For good performance, *lwork* must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork

REAL for *cgges*

DOUBLE PRECISION for *zgges*

Workspace array, size at least $\max(1, 8n)$.

This array is used in complex flavors only.

bwork

LOGICAL.

Workspace array, size at least $\max(1, n)$.

Not referenced if *sort* = 'N'.

Output Parameters

a

On exit, this array has been overwritten by its generalized Schur form *S*.

b

On exit, this array has been overwritten by its generalized Schur form *T*.

sdim

INTEGER.

If *sort* = 'N', *sdim* = 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selectg* is true.

Note that for real flavors complex conjugate pairs for which *selectg* is true for either eigenvalue count as 2.

alphan, *alphai*

REAL for *sgges*;

DOUBLE PRECISION for *dgges*.

Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for *cgges*;

DOUBLE COMPLEX for *zgges*.

Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for *sgges*

DOUBLE PRECISION for *dgges*

COMPLEX for *cgges*

DOUBLE COMPLEX for *zgges*.

Array, size at least $\max(1, n)$.

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

$\text{alphar}(j) + \text{alphai}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations. If $\text{alphai}(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\text{alphai}(j+1)$ negative.

For complex flavors:

On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues. $\text{alpha}(j)$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) output by *cgges/zgges*. The $\text{beta}(j)$ will be non-negative real.

See also *Application Notes* below.

vs1, vsr

REAL for *sgges*

DOUBLE PRECISION for *dgges*

COMPLEX for *cgges*

DOUBLE COMPLEX for *zgges*.

Arrays:

vs1(*ldvs1*,*), the second dimension of *vs1* must be at least $\max(1, n)$.

If *jobvs1* = 'V', this array will contain the left Schur vectors.

If *jobvs1* = 'N', *vs1* is not referenced.

vsr(*ldvsr*,*), the second dimension of *vsr* must be at least $\max(1, n)$.

If *jobvsr* = 'V', this array will contain the right Schur vectors.

If *jobvsr* = 'N', *vsr* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

$i \leq n$:

the QZ iteration failed. (A, B) is not in Schur form, but $\text{alphar}(j)$, $\text{alphai}(j)$ (for real flavors), or $\text{alpha}(j)$ (for complex flavors), and $\text{beta}(j)$, $j=\text{info}+1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in *hgeqz*;

$i = n+2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy *selctg* = .TRUE.. This could also be caused due to scaling;

$i = n+3$: reordering failed in `tgsen`.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `gges` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size (n, n) .
<code>alphar</code>	Holds the vector of length n . Used in real flavors only.
<code>alphai</code>	Holds the vector of length n . Used in real flavors only.
<code>alpha</code>	Holds the vector of length n . Used in complex flavors only.
<code>beta</code>	Holds the vector of length n .
<code>vsl</code>	Holds the matrix VSL of size (n, n) .
<code>vsr</code>	Holds the matrix VSR of size (n, n) .
<code>jobvsl</code>	Restored based on the presence of the argument <code>vsl</code> as follows: <code>jobvsl = 'V'</code> , if <code>vsl</code> is present, <code>jobvsl = 'N'</code> , if <code>vsl</code> is omitted.
<code>jobvsr</code>	Restored based on the presence of the argument <code>vsr</code> as follows: <code>jobvsr = 'V'</code> , if <code>vsr</code> is present, <code>jobvsr = 'N'</code> , if <code>vsr</code> is omitted.
<code>sort</code>	Restored based on the presence of the argument <code>select</code> as follows: <code>sort = 'S'</code> , if <code>select</code> is present, <code>sort = 'N'</code> , if <code>select</code> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients $\alpha(j)/\beta(j)$ and $\alpha_i(j)/\beta(j)$ may easily over- or underflow, and $\beta(j)$ may even be zero. Thus, you should avoid simply computing the ratio. However, α and α_i will be always less than and usually comparable with $\text{norm}(A)$ in magnitude, and β always less than and usually comparable with $\text{norm}(B)$.

sggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphas,
alpha_i, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork,
bwork, info)
```

```
call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphas,
alpha_i, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork,
bwork, info)
```

```
call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha,
beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork,
bwork, info)
```

```
call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha,
beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork,
bwork, info)
```

```
call ggesx(a, b, alphas, alpha_i, beta [,vsl] [,vsr] [,select] [,sdim] [,rconde] [,
rcondv] [,info])
```

```
call ggesx(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,rconde] [,rcondv] [,
info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A,B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T ; computes a reciprocal condition number for the average of the selected eigenvalues ($rconde$); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues ($rcondv$). The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $\alpha / \beta = w$, such that $A - w * B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta=0$ or for both being zero. A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

Input Parameters

<i>jobvsl</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvsl</i> = 'N', then the left Schur vectors are not computed.</p> <p>If <i>jobvsl</i> = 'V', then the left Schur vectors are computed.</p>
<i>jobvsr</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvsr</i> = 'N', then the right Schur vectors are not computed.</p> <p>If <i>jobvsr</i> = 'V', then the right Schur vectors are computed.</p>
<i>sort</i>	<p>CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.</p> <p>If <i>sort</i> = 'N', then eigenvalues are not ordered.</p> <p>If <i>sort</i> = 'S', eigenvalues are ordered (see <i>selctg</i>).</p>
<i>selctg</i>	<p>LOGICAL FUNCTION of three REAL arguments for real flavors.</p> <p>LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.</p> <p><i>selctg</i> must be declared EXTERNAL in the calling subroutine.</p> <p>If <i>sort</i> = 'S', <i>selctg</i> is used to select eigenvalues to sort to the top left of the Schur form.</p> <p>If <i>sort</i> = 'N', <i>selctg</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>An eigenvalue $(\text{alphan}(j) + \text{alphai}(j))/\text{betan}(j)$ is selected if <i>selctg</i>(<i>alphan</i>(<i>j</i>), <i>alphai</i>(<i>j</i>), <i>betan</i>(<i>j</i>)) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p>Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alphan</i>(<i>j</i>), <i>alphai</i>(<i>j</i>), <i>betan</i>(<i>j</i>)) = .TRUE. after ordering. In this case <i>info</i> is set to <i>n</i>+2.</p> <p><i>For complex flavors:</i></p> <p>An eigenvalue $\text{alpha}(j) / \text{betan}(j)$ is selected if <i>selctg</i>(<i>alpha</i>(<i>j</i>), <i>betan</i>(<i>j</i>)) is true.</p> <p>Note that a selected complex eigenvalue may no longer satisfy <i>selctg</i>(<i>alpha</i>(<i>j</i>), <i>betan</i>(<i>j</i>)) = .TRUE. after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> is set to <i>n</i>+2 (see <i>info</i> below).</p>
<i>sense</i>	<p>CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p>

If *sense* = 'N', none are computed;
 If *sense* = 'E', computed for average of selected eigenvalues only;
 If *sense* = 'V', computed for selected deflating subspaces only;
 If *sense* = 'B', computed for both.
 If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n INTEGER. The order of the matrices *A*, *B*, *vsl*, and *vsr* ($n \geq 0$).

a, *b*, *work* REAL for sggex
 DOUBLE PRECISION for dggex
 COMPLEX for cggex
 DOUBLE COMPLEX for zggex.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least $\max(1, n)$.

b(*ldb*,*) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array *a*.
 Must be at least $\max(1, n)$.

ldb INTEGER. The leading dimension of the array *b*.
 Must be at least $\max(1, n)$.

ldvsl, *ldvsr* INTEGER. The leading dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:
ldvsl ≥ 1 . If *jobvsl* = 'V', *ldvsl* $\geq \max(1, n)$.
ldvsr ≥ 1 . If *jobvsr* = 'V', *ldvsr* $\geq \max(1, n)$.

lwork INTEGER.
 The dimension of the array *work*.
For real flavors:
 If $n=0$ then *lwork* ≥ 1 .
 If $n>0$ and *sense* = 'N', then *lwork* $\geq \max(8*n, 6*n+16)$.
 If $n>0$ and *sense* = 'E', 'V', or 'B', then *lwork* $\geq \max(8*n, 6*n+16, 2*sdim*(n-sdim))$;
For complex flavors:
 If $n=0$ then *lwork* ≥ 1 .
 If $n>0$ and *sense* = 'N', then *lwork* $\geq \max(1, 2*n)$;

If $n > 0$ and $sense = 'E', 'V',$ or $'B'$, then $lwork \geq \max(1, 2*n, 2*sdim*(n-sdim))$.

Note that $2*sdim*(n-sdim) \leq n*n/2$.

An error is only returned if $lwork < \max(8*n, 6*n+16)$ for real flavors, and $lwork < \max(1, 2*n)$ for complex flavors, but if $sense = 'E', 'V',$ or $'B'$, this may not be large enough.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

rwork

REAL for cggex

DOUBLE PRECISION for zggex

Workspace array, size at least $\max(1, 8n)$.

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, size $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

If $sense = 'N'$, or $n = 0$, then $liwork \geq 1$,

otherwise $liwork \geq (n+6)$ for real flavors, and $liwork \geq (n+2)$ for complex flavors.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

bwork

LOGICAL.

Workspace array, size at least $\max(1, n)$.

Not referenced if $sort = 'N'$.

Output Parameters

a

On exit, this array has been overwritten by its generalized Schur form *S*.

b

On exit, this array has been overwritten by its generalized Schur form *T*.

sdim

INTEGER.

If $sort = 'N'$, $sdim = 0$.

If $sort = 'S'$, *sdim* is equal to the number of eigenvalues (after sorting) for which *selctg* is true.

Note that for real flavors complex conjugate pairs for which *selectg* is true for either eigenvalue count as 2.

alphar, alphas

REAL for *sggesx*;
DOUBLE PRECISION for *dggesx*.

Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for *cggesx*;
DOUBLE COMPLEX for *zggesx*.

Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for *sggesx*
DOUBLE PRECISION for *dggesx*
COMPLEX for *cggesx*
DOUBLE COMPLEX for *zggesx*.

Array, size at least $\max(1, n)$.

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$ will be the generalized eigenvalues.

$\text{alphar}(j) + \text{alphai}(j)*i$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations. If $\text{alphai}(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\text{alphai}(j+1)$ negative.

For complex flavors:

On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$ will be the generalized eigenvalues. $\text{alpha}(j)$ and $\text{beta}(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) output by *cggesx/zggesx*. The $\text{beta}(j)$ will be non-negative real.

See also *Application Notes* below.

vsr, vsr

REAL for *sggesx*
DOUBLE PRECISION for *dggesx*
COMPLEX for *cggesx*
DOUBLE COMPLEX for *zggesx*.

Arrays:

$\text{vsr}(\text{ldvsr}, *)$, the second dimension of *vsr* must be at least $\max(1, n)$.

If $\text{jobvsr} = 'V'$, this array will contain the left Schur vectors.

If $\text{jobvsr} = 'N'$, *vsr* is not referenced.

$\text{vsr}(\text{ldvsr}, *)$, the second dimension of *vsr* must be at least $\max(1, n)$.

If $\text{jobvsr} = 'V'$, this array will contain the right Schur vectors.

	If $jobvsr = 'N'$, vsr is not referenced.
$rconde, rcondv$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, size (2) each If $sense = 'E'$ or $'B'$, $rconde[0]$ and $rconde[1]$ contain the reciprocal condition numbers for the average of the selected eigenvalues. Not referenced if $sense = 'N'$ or $'V'$. If $sense = 'V'$ or $'B'$, $rcondv(1)$ and $rcondv(2)$ contain the reciprocal condition numbers for the selected deflating subspaces. Not referenced if $sense = 'N'$ or $'E'$.
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i th parameter had an illegal value. If $info = i$, and $i \leq n$: the QZ iteration failed. (A, B) is not in Schur form, but $alphar(j)$, $alphai(j)$ (for real flavors), or $alpha(j)$ (for complex flavors), and $beta(j)$, $j=info+1, \dots, n$ should be correct. $i > n$: errors that usually indicate LAPACK problems: $i = n+1$: other than QZ iteration failed in ?hgeqz; $i = n+2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy $selctg = .TRUE..$ This could also be caused due to scaling; $i = n+3$: reordering failed in tgsen .

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ggesx` interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
$alphar$	Holds the vector of length n . Used in real flavors only.
$alphai$	Holds the vector of length n . Used in real flavors only.

<i>alpha</i>	Holds the vector of length n . Used in complex flavors only.
<i>beta</i>	Holds the vector of length n .
<i>vs1</i>	Holds the matrix <i>VSL</i> of size (n, n) .
<i>vsr</i>	Holds the matrix <i>VSR</i> of size (n, n) .
<i>rconde</i>	Holds the vector of length (2).
<i>rcondv</i>	Holds the vector of length (2).
<i>jobvs1</i>	Restored based on the presence of the argument <i>vs1</i> as follows: <i>jobvs1</i> = 'V', if <i>vs1</i> is present, <i>jobvs1</i> = 'N', if <i>vs1</i> is omitted.
<i>jobvsr</i>	Restored based on the presence of the argument <i>vsr</i> as follows: <i>jobvsr</i> = 'V', if <i>vsr</i> is present, <i>jobvsr</i> = 'N', if <i>vsr</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present, <i>sort</i> = 'N', if <i>select</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Note that there will be an error condition if *rconde* or *rcondv* are present and *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar*(j)/*beta*(j) and *alphai*(j)/*beta*(j) may easily over- or underflow, and *beta*(j) may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm*(*A*) in magnitude, and *beta* always less than and usually comparable with *norm*(*B*).

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

```
call sggev(jobvl, jobvr, n, a, lda, b, ldb, alphas, alphai, beta, vl, ldvl, vr, ldvr,
work, lwork, info)
```

```
call dggev(jobvl, jobvr, n, a, lda, b, ldb, alphas, alphai, beta, vl, ldvl, vr, ldvr,
work, lwork, info)
```

```
call cggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work,
lwork, rwork, info)
```

```
call zggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work,
lwork, rwork, info)
```

```
call ggev(a, b, alphas, alphai, beta [,vl] [,vr] [,info])
```

```
call ggev(a, b, alpha, beta [, vl] [,vr] [,info])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?ggev routine computes the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors for a pair of n -by- n real/complex nonsymmetric matrices (A,B) .

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $alpha / beta = \lambda$, such that $A - \lambda*B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta = 0$ and even for both being zero.

The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies $A*v(j) = \lambda(j)*B*v(j)$.

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies $u(j)^H*A = \lambda(j)*u(j)^H*B$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

The ?ggev routine replaces the deprecated ?gegv routine.

Input Parameters

<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed; If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.
<i>jobvr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed; If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrices <i>A</i> , <i>B</i> , <i>vl</i> , and <i>vr</i> ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for sggev

DOUBLE PRECISION for dggev

COMPLEX for cggev

DOUBLE COMPLEX for zggev.

Arrays:

$a(lda,*)$ is an array containing the n -by- n matrix A (first of the pair of matrices).

The second dimension of a must be at least $\max(1, n)$.

$b(ldb,*)$ is an array containing the n -by- n matrix B (second of the pair of matrices).

The second dimension of b must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

ldb INTEGER. The leading dimension of the array b . Must be at least $\max(1, n)$.

$ldvl, ldvr$ INTEGER. The leading dimensions of the output matrices v_l and v_r , respectively.

Constraints:

$ldvl \geq 1$. If $jobvl = 'V'$, $ldvl \geq \max(1, n)$.

$ldvr \geq 1$. If $jobvr = 'V'$, $ldvr \geq \max(1, n)$.

$lwork$ INTEGER.

The dimension of the array $work$.

$lwork \geq \max(1, 8n+16)$ for real flavors;

$lwork \geq \max(1, 2n)$ for complex flavors.

For good performance, $lwork$ must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

$rwork$ REAL for cggev

DOUBLE PRECISION for zggev

Workspace array, size at least $\max(1, 8n)$.

This array is used in complex flavors only.

Output Parameters

a, b On exit, these arrays have been overwritten.

$alphar, alphas$ REAL for sggev;

DOUBLE PRECISION for dggev.

Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha COMPLEX for *cggev*;
 DOUBLE COMPLEX for *zggev*.
 Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta REAL for *sggev*
 DOUBLE PRECISION for *dggev*
 COMPLEX for *cggev*
 DOUBLE COMPLEX for *zggev*.
 Array, size at least $\max(1, n)$.
 For real flavors:
 On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, are the generalized eigenvalues.
 If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(*j*+1) negative.
 For complex flavors:
 On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$, are the generalized eigenvalues.
 See also *Application Notes* below.

vl, vr REAL for *sggev*
 DOUBLE PRECISION for *dggev*
 COMPLEX for *cggev*
 DOUBLE COMPLEX for *zggev*.
 Arrays:
vl(*ldvl*,*); the second dimension of *vl* must be at least $\max(1, n)$. Contains the matrix of left generalized eigenvectors *VL*.
 If *jobvl* = 'V', the left generalized eigenvectors u_j are stored one after another in the columns of *VL*, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.
 If *jobvl* = 'N', *vl* is not referenced.
 For real flavors:
 If the *j*-th eigenvalue is real, then $u_j = \text{VL}_{*,j}$, the *j*-th column of *VL*.
 If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \sqrt{-1}$, $u_j = \text{VL}_{*,j} + i*\text{VL}_{*,j+1}$ and $u_{j+1} = \text{VL}_{*,j} - i*\text{VL}_{*,j+1}$.
 For complex flavors:
 $u_j = \text{VL}_{*,j}$, the *j*-th column of *vl*.
vr(*ldvr*,*); the second dimension of *vr* must be at least $\max(1, n)$. Contains the matrix of right generalized eigenvectors *VR*.

If $jobvnr = 'V'$, the right generalized eigenvectors v_j are stored one after another in the columns of VR , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $abs(Re) + abs(Im) = 1$.

If $jobvnr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v_j = VR_{*,j}$, the j -th column of VR .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v_j = VR_{*,j} + i*VR_{*,j+1}$ and $v_{j+1} = VR_{*,j} - i*VR_{*,j+1}$.

For complex flavors:

$v_j = VR_{*,j}$, the j -th column of VR .

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and

$i \leq n$: the QZ iteration failed. No eigenvectors have been calculated, but $alpha(j)$, $alpha_i(j)$ (for real flavors), or $alpha(j)$ (for complex flavors), and $beta(j)$, $j=info+1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in [hgeqz](#);

$i = n+2$: error return from [tgevc](#).

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ggevc` interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
$alpha_r$	Holds the vector of length n . Used in real flavors only.
$alpha_i$	Holds the vector of length n . Used in real flavors only.
$alpha$	Holds the vector of length n . Used in complex flavors only.
$beta$	Holds the vector of length n .
vl	Holds the matrix VL of size (n, n) .
vr	Holds the matrix VR of size (n, n) .
$jobv1$	Restored based on the presence of the argument vl as follows:

$jobvl = 'V'$, if vl is present,

$jobvl = 'N'$, if vl is omitted.

$jobvr$

Restored based on the presence of the argument vr as follows:

$jobvr = 'V'$, if vr is present,

$jobvr = 'N'$, if vr is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients $alphan(j)/beta(j)$ and $alphai(j)/beta(j)$ may easily over- or underflow, and $beta(j)$ may even be zero. Thus, you should avoid simply computing the ratio. However, $alphan$ and $alphai$ (for real flavors) or $alpha$ (for complex flavors) will be always less than and usually comparable with $\text{norm}(A)$ in magnitude, and $beta$ always less than and usually comparable with $\text{norm}(B)$.

?ggevx

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

```
call sggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphan, alphai, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork,
iwork, bwork, info)
```

```
call dggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphan, alphai, beta, vl,
ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork,
iwork, bwork, info)
```

```
call cggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr,
ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, rwork,
iwork, bwork, info)
```

```
call zggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr,
ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, rwork,
iwork, bwork, info)
```

```
call ggevx(a, b, alphan, alphai, beta [,vl] [,vr] [,balanc] [,ilo] [,ihi] [, lscale]
[,rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

```
call ggevx(a, b, alpha, beta [, vl] [,vr] [,balanc] [,ilo] [,ihi] [,lscale] [, rscale]
[,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $alpha / beta = \lambda$, such that $A - \lambda*B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta=0$ and even for both being zero. The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$A*v(j) = \lambda(j)*B*v(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$u(j)^{H*A} = \lambda(j)*u(j)^{H*B}$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

<i>balanc</i>	<p>CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed.</p> <p>If <i>balanc</i> = 'N', do not diagonally scale or permute;</p> <p>If <i>balanc</i> = 'P', permute only;</p> <p>If <i>balanc</i> = 'S', scale only;</p> <p>If <i>balanc</i> = 'B', both permute and scale.</p> <p>Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.</p>
<i>jobvl</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed;</p> <p>If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.</p>
<i>jobvr</i>	<p>CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed;</p> <p>If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.</p>
<i>sense</i>	<p>CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = 'N', none are computed;</p> <p>If <i>sense</i> = 'E', computed for eigenvalues only;</p> <p>If <i>sense</i> = 'V', computed for eigenvectors only;</p> <p>If <i>sense</i> = 'B', computed for eigenvalues and eigenvectors.</p>

<i>n</i>	INTEGER. The order of the matrices <i>A</i> , <i>B</i> , <i>vl</i> , and <i>vr</i> ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for <i>sggev</i> DOUBLE PRECISION for <i>dggev</i> COMPLEX for <i>cggev</i> DOUBLE COMPLEX for <i>zggev</i> . Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> (first of the pair of matrices). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>B</i> (second of the pair of matrices). The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . Must be at least $\max(1, n)$.
<i>ldvl</i> , <i>ldvr</i>	INTEGER. The leading dimensions of the output matrices <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1 . If <i>jobvl</i> = 'V', <i>ldvl</i> $\geq \max(1, n)$. <i>ldvr</i> ≥ 1 . If <i>jobvr</i> = 'V', <i>ldvr</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> $\geq \max(1, 2*n)$; For real flavors: If <i>balanc</i> = 'S', or 'B', or <i>jobvl</i> = 'V', or <i>jobvr</i> = 'V', then <i>lwork</i> $\geq \max(1, 6*n)$; if <i>sense</i> = 'E', or 'B', then <i>lwork</i> $\geq \max(1, 10*n)$; if <i>sense</i> = 'V', or 'B', <i>lwork</i> $\geq (2n^2 + 8*n + 16)$. For complex flavors: if <i>sense</i> = 'E', <i>lwork</i> $\geq \max(1, 4*n)$; if <i>sense</i> = 'V', or 'B', <i>lwork</i> $\geq \max(1, 2*n^2 + 2*n)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla .
<i>rwork</i>	REAL for <i>cggev</i> DOUBLE PRECISION for <i>zggev</i>

Workspace array, size at least $\max(1, 6*n)$ if *balanc* = 'S', or 'B', and at least $\max(1, 2*n)$ otherwise.

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, size at least $(n+6)$ for real flavors and at least $(n+2)$ for complex flavors.

Not referenced if *sense* = 'E'.

bwork

LOGICAL. Workspace array, size at least $\max(1, n)$.

Not referenced if *sense* = 'N'.

Output Parameters

a, b

On exit, these arrays have been overwritten.

If *jobvl* = 'V' or *jobvr* = 'V' or both, then *a* contains the first part of the real Schur form of the "balanced" versions of the input *A* and *B*, and *b* contains its second part.

alphar, alphas

REAL for *sggevx*;

DOUBLE PRECISION for *dggevx*.

Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for *cggevx*;

DOUBLE COMPLEX for *zggevx*.

Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for *sggevx*

DOUBLE PRECISION for *dggevx*

COMPLEX for *cggevx*

DOUBLE COMPLEX for *zggevx*.

Array, size at least $\max(1, n)$.

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-st eigenvalues are a complex conjugate pair, with *alphai*(*j*+1) negative.

For complex flavors:

On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

See also *Application Notes* below.

vl, vr

REAL for *sggevx*

DOUBLE PRECISION for `dggev`

COMPLEX for `cggev`

DOUBLE COMPLEX for `zggev`.

Arrays:

$vl(ldvl,*)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:, j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, $u(j) = vl(:, j) + i*vl(:, j+1)$ and $u(j+1) = vl(:, j) - i*vl(:, j+1)$.

For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .

$vr(ldvr,*)$; the second dimension of vr must be at least $\max(1, n)$.

If $jobvvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i*vr(:, j+1)$ and $v(j+1) = vr(:, j) - i*vr(:, j+1)$.

For complex flavors:

$v(j) = vr(:, j)$, the j -th column of vr .

ilo, ihi

INTEGER. ilo and ihi are integer values such that on exit $A_{ij} = 0$ and $B_{ij} = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

$lscale, rscale$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, size at least $\max(1, n)$ each.

$lscale$ contains details of the permutations and scaling factors applied to the left side of A and B .

If $PL(j)$ is the index of the row interchanged with row j , and $DL(j)$ is the scaling factor applied to row j , then

$lscale(j) = PL(j)$, for $j = 1, \dots, ilo-1$
 $= DL(j)$, for $j = ilo, \dots, ihi$
 $= PL(j)$ for $j = ihi+1, \dots, n$.

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

rscal contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If $PR(j)$ is the index of the column interchanged with column j , and $DR(j)$ is the scaling factor applied to column j , then

$rscal(j) = PR(j)$, for $j = 1, \dots, ilo-1$
 $= DR(j)$, for $j = ilo, \dots, ihi$
 $= PR(j)$ for $j = ihi+1, \dots, n$.

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

abnrm, bbnrm

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norms of the balanced matrices *A* and *B*, respectively.

rconde, rcondv

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(1, n)$ each.

If *sense* = 'E', or 'B', *rconde* contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde(j)*, *rcondv(j)*, and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*-th eigenpair, unless all eigenpairs are selected).

If *sense* = 'N', or 'V', *rconde* is not referenced.

If *sense* = 'V', or 'B', *rcondv* contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value.

If the eigenvalues cannot be reordered to compute *rcondv(j)rconde[j]*, *rcondv(j)* is set to 0; this can only occur when the true value would be very small anyway.

If *sense* = 'N', or 'E', *rcondv* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

$i \leq n$:

the QZ iteration failed. No eigenvectors have been calculated, but $alphan(j)$, $alpha_i(j)$ (for real flavors), or $alpha(j)$ (for complex flavors), and $beta(j)$, $j=info+1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in `hgeqz`;

$i = n+2$: error return from `tgevc`.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `ggevx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size (n, n) .
<i>alphan</i>	Holds the vector of length n . Used in real flavors only.
<i>alpha_i</i>	Holds the vector of length n . Used in real flavors only.
<i>alpha</i>	Holds the vector of length n . Used in complex flavors only.
<i>beta</i>	Holds the vector of length n .
<i>vl</i>	Holds the matrix <i>VL</i> of size (n, n) .
<i>vr</i>	Holds the matrix <i>VR</i> of size (n, n) .
<i>lscale</i>	Holds the vector of length n .
<i>rscale</i>	Holds the vector of length n .
<i>rconde</i>	Holds the vector of length n .
<i>rcondv</i>	Holds the vector of length n .
<i>balanc</i>	Must be 'N', 'B', or 'P'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present,

$sense = 'N'$, if both $rconde$ and $rcondv$ are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients $alphar(j)/beta(j)$ and $alphai(j)/beta(j)$ may easily over- or underflow, and $beta(j)$ may even be zero. Thus, you should avoid simply computing the ratio. However, $alphar$ and $alphai$ (for real flavors) or $alpha$ (for complex flavors) will be always less than and usually comparable with $norm(A)$ in magnitude, and $beta$ always less than and usually comparable with $norm(B)$.

Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?syconv	s, c, d, z	Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.

Routine Name	Data Types	Description
i?max1	<i>c, z</i>	Finds the index of the vector element whose real part has maximum absolute value.
?sum1	<i>sc, dz</i>	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	<i>s, d, c, z</i>	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.
?gebd2	<i>s, d, c, z</i>	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	<i>s, d, c, z</i>	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	<i>s, d, c, z</i>	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?geql2	<i>s, d, c, z</i>	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	<i>s, d, c, z</i>	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2p	<i>s, d, c, z</i>	Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.
?geqrt2	<i>s, d, c, z</i>	Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?geqrt3	<i>s, d, c, z</i>	Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?gerq2	<i>s, d, c, z</i>	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	<i>s, d, c, z</i>	Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2 .
?getc2	<i>s, d, c, z</i>	Computes the LU factorization with complete pivoting of the general n -by- n matrix.
?getf2	<i>s, d, c, z</i>	Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	<i>s, d, c, z</i>	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf .
?isnan	<i>s, d,</i>	Tests input for NaN.
?laisnan	<i>s, d,</i>	Tests input for NaN by comparing two arguments for inequality.
?labrd	<i>s, d, c, z</i>	Reduces the first nb rows and columns of a general matrix to a bidiagonal form.
?lacn2	<i>s, d, c, z</i>	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacon	<i>s, d, c, z</i>	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	<i>s, d, c, z</i>	Copies all or part of one two-dimensional array to another.

Routine Name	Data Types	Description
?ladiv	s, d, c, z	Performs complex division in real arithmetic, avoiding unnecessary overflow.
?lae2	s, d	Computes the eigenvalues of a 2-by-2 symmetric matrix.
?laebz	s, d	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.
?laed0	s, d, c, z	Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
?laed1	s, d	Used by sstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
?laed2	s, d	Used by sstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
?laed3	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.
?laed4	s, d	Used by sstedc/dstedc. Finds a single root of the secular equation.
?laed5	s, d	Used by sstedc/dstedc. Solves the 2-by-2 secular equation.
?laed6	s, d	Used by sstedc/dstedc. Computes one Newton step in solution of the secular equation.
?laed7	s, d, c, z	Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.
?laed8	s, d, c, z	Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
?laed9	s, d	Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
?laeda	s, d	Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
?laein	s, d, c, z	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
?laev2	s, d, c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s, d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Routine Name	Data Types	Description
?lags2	s, d	Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.
?lagtf	s, d	Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
?lagtm	s, d, c, z	Performs a matrix-matrix product of the form $C = \alpha AB + \beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.
?lagts	s, d	Solves the system of equations $(T-\lambda I)x = y$ or $(T-\lambda I)^T x = y$, where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf.
?lagv2	s, d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A, B) where B is upper triangular.
?lahqr	s, d, c, z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?lahr2	s, d, c, z	Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?laic1	s, d, c, z	Applies one step of incremental condition estimation.
?lakf2	s, d, c, z	Forms a matrix containing Kronecker products between the given matrices.
?lals0	s, d, c, z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.
?lalsa	s, d, c, z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.
?lalsd	s, d, c, z	Uses the singular value decomposition of A to solve the least squares problem.
?lamrg	s, d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.
?laneg	s, d	Computes the Sturm count.
?langb	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Routine Name	Data Types	Description
<code>?langt</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
<code>?lanhs</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
<code>?lansb</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.
<code>?lanhb</code>	<code>c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
<code>?lansp</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.
<code>?lanhp</code>	<code>c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
<code>?lanst/?lanht</code>	<code>s, d/c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.
<code>?lansy</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.
<code>?lanhe</code>	<code>c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.
<code>?lantb</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
<code>?lantp</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
<code>?lantr</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
<code>?lanv2</code>	<code>s, d</code>	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
<code>?lap11</code>	<code>s, d, c, z</code>	Measures the linear dependence of two vectors.
<code>?lapmr</code>	<code>s, d, c, z</code>	Rearranges rows of a matrix as specified by a permutation vector.
<code>?lapmt</code>	<code>s, d, c, z</code>	Performs a forward or backward permutation of the columns of a matrix.
<code>?lapy2</code>	<code>s, d</code>	Returns $\sqrt{x^2+y^2}$.
<code>?lapy3</code>	<code>s, d</code>	Returns $\sqrt{x^2+y^2+z^2}$.

Routine Name	Data Types	Description
?laqgb	s, d, c, z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ.
?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.
?laqhb	c, z	Scales a Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqp2	s, d, c, z	Computes a QR factorization with column pivoting of the matrix block.
?laqps	s, d, c, z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix A by using BLAS level 3.
?laqr0	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr1	s, d, c, z	Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix H and specified shifts.
?laqr2	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr3	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr4	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr5	s, d, c, z	Performs a single small-bulge multi-shift QR sweep.
?laqsb	s, d, c, z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqsp	s, d, c, z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.
?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.
?laqtr	s, d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
?lar1v	s, d, c, z	Computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $LDL^T - \lambda I$.
?lar2v	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?laran	s, d	Returns a random real number from a uniform distribution.
?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).

Routine Name	Data Types	Description
?larfgp	s, d, c, z	Generates an elementary reflector (Householder matrix) with non-negative beta.
?larft	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - \alpha v v^H$
?larfx	s, d, c, z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10 .
?large	s, d, c, z	Pre- and post-multiplies a real general matrix with a random orthogonal matrix.
?larnd	s, d, c, z	Returns a random real number from a uniform or normal distribution.
?largv	s, d, c, z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s, d, c, z	Returns a vector of random numbers from a uniform or normal distribution.
?laror	s, d, c, z	Pre- or post-multiplies an m -by- n matrix by a random orthogonal/unitary matrix.
?larot	s, d, c, z	Applies a Givens rotation to two adjacent rows or columns.
?larra	s, d	Computes the splitting points with the specified threshold.
?larrb	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrc	s, d	Computes the number of eigenvalues of the symmetric tridiagonal matrix.
?larrd	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre	s, d	Given the tridiagonal matrix T , sets small off-diagonal elements to zero and for each unreduced block T_i , finds base representations and eigenvalues.
?larrf	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrj	s, d	Performs refinement of the initial estimates of the eigenvalues of the matrix T .
?larrk	s, d	Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.
?larrl	s, d	Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.
?larrv	s, d, c, z	Computes the eigenvectors of the tridiagonal matrix $T = LDL^T$ given L , D and the eigenvalues of LDL^T .
?lartg	s, d, c, z	Generates a plane rotation with real cosine and real/complex sine.
?lartgp	s, d	Generates a plane rotation so that the diagonal is nonnegative.
?lartgs	s, d	Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Routine Name	Data Types	Description
?lartv	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
?laruv	s, d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s, d, c, z	Applies an elementary reflector (as returned by ?tzzrf) to a general matrix.
?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?las2	s, d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .
?lasd0	s, d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasd1	s, d	Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.
?lasd2	s, d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.
?lasd3	s, d	Finds all square roots of the roots of the secular equation, as defined by the values in D and Z , and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.
?lasd4	s, d	Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.
?lasd5	s, d	Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.
?lasd6	s, d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.
?lasd7	s, d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.
?lasd8	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc.
?lasda	s, d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasdq	s, d	Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasdt	s, d	Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Routine Name	Data Types	Description
?laset	s, d, c, z	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.
?lasq1	s, d	Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.
?lasq2	s, d	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the <i>qd</i> Array <i>Z</i> to high relative accuracy. Used by ?bdsqr and ?stegr.
?lasq3	s, d	Checks for deflation, computes a shift and calls <i>dqds</i> . Used by ?bdsqr.
?lasq4	s, d	Computes an approximation to the smallest eigenvalue using values of <i>d</i> from the previous transform. Used by ?bdsqr.
?lasq5	s, d	Computes one <i>dqds</i> transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasq6	s, d	Computes one <i>dqd</i> transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasr	s, d, c, z	Applies a sequence of plane rotations to a general rectangular matrix.
?lasrt	s, d	Sorts numbers in increasing or decreasing order.
?lassq	s, d, c, z	Updates a sum of squares represented in scaled form.
?lasv2	s, d	Computes the singular value decomposition of a 2-by-2 triangular matrix.
?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
?lasy2	s, d	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.
?lasyf	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lasyf_rook	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?lahef	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?lahef_rook	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?latbs	s, d, c, z	Solves a triangular banded system of equations.
?latdf	s, d, c, z	Uses the LU factorization of the <i>n</i> -by- <i>n</i> matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latm1	s, d, c, z	Computes the entries of a matrix as specified.
?latm2	s, d, c, z	Returns an entry of a random matrix.
?latm3	s, d, c, z	Returns set entry of a random matrix.

Routine Name	Data Types	Description
?latm5	s, d, c, z	Generates matrices involved in the Generalized Sylvester equation.
?latm6	s, d, c, z	Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.
?latme	s, d, c, z	Generates random non-symmetric square matrices with specified eigenvalues.
?latmr	s, d, c, z	Generates random matrices of various types.
?latps	s, d, c, z	Solves a triangular system of equations with the matrix held in packed storage.
?latrd	s, d, c, z	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s, d, c, z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s, d, c, z	Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.
?lauu2	s, d, c, z	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices (unblocked algorithm).
?lauum	s, d, c, z	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices (blocked algorithm).
?orbdb1/?unbdb1	s, d, c, z	Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
?orbdb2/?unbdb2		
?orbdb3/?unbdb3		
?orbdb4/?unbdb4		
?orbdb5/?unbdb5	s, d, c, z	Orthogonalizes a column vector with respect to the orthonormal basis matrix.
?orbdb6/?unbdb6		
?org2l/?ung2l	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).
?org2r/?ung2r	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).
?orgr2/?ungr2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).

Routine Name	Data Types	Description
?orml2/?unml2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).
?ormr2/?unmr2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).
?ormr3/?unmr3	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).
?pbt2	s, d, c, z	Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).
?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s, d, c, z	Solves a tridiagonal system of the form $AX=B$ using the LDL^H factorization computed by ?pttrf .
?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?syswapr	s, d, c, z	Applies an elementary permutation on the rows and columns of a symmetric matrix.
?heswapr	c, z	Applies an elementary permutation on the rows and columns of a Hermitian matrix.
?sygs2/?hegs2	s, d/c, z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s, d/c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?sytf2_rook	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?hetf2	c, z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).
?hetf2_rook	c, z	Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?tgex2	s, d, c, z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.
?tgsy2	s, d, c, z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s, d, c, z	Computes the inverse of a triangular matrix (unblocked algorithm).
clag2z	c→z	Converts a complex single precision matrix to a complex double precision matrix.
dlag2s	d→s	Converts a double precision matrix to a single precision matrix.

Routine Name	Data Types	Description
<code>slag2d</code>	<code>s→d</code>	Converts a single precision matrix to a double precision matrix.
<code>zlag2c</code>	<code>z→c</code>	Converts a complex double precision matrix to a complex single precision matrix.
<code>?larfp</code>	<code>s, d, c, z</code>	Generates a real or complex elementary reflector.
<code>ila?lc</code>	<code>s, d, c, z</code>	Scans a matrix for its last non-zero column.
<code>ila?lr</code>	<code>s, d, c, z</code>	Scans a matrix for its last non-zero row.
<code>?gsvj0</code>	<code>s, d</code>	Pre-processor for the routine <code>?gesvj</code> .
<code>?gsvj1</code>	<code>s, d</code>	Pre-processor for the routine <code>?gesvj</code> , applies Jacobi rotations targeting only particular pivots.
<code>?sfrk</code>	<code>s, d</code>	Performs a symmetric rank-k operation for matrix in RFP format.
<code>?hfrk</code>	<code>c, z</code>	Performs a Hermitian rank-k operation for matrix in RFP format.
<code>?tfsm</code>	<code>s, d, c, z</code>	Solves a matrix equation (one operand is a triangular matrix in RFP format).
<code>?lansf</code>	<code>s, d</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.
<code>?lanhf</code>	<code>c, z</code>	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.
<code>?tfttp</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).
<code>?tfttr</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR).
<code>?tpqrt2</code>	<code>s, d, c, z</code>	Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.
<code>?tprfb</code>	<code>s, d, c, z</code>	Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.
<code>?tpttf</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).
<code>?tpttr</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).
<code>?trttf</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).
<code>?trttp</code>	<code>s, d, c, z</code>	Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).
<code>?pstf2</code>	<code>s, d, c, z</code>	Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.
<code>dlat2s</code>	<code>d→s</code>	Converts a double-precision triangular matrix to a single-precision triangular matrix.

Routine Name	Data Types	Description
zlat2c	$z \rightarrow c$	Converts a double complex triangular matrix to a complex triangular matrix.
?laccp2	c, z	Copies all or part of a real two-dimensional array to a complex array.
?la_gbamv	s, d, c, z	Performs a matrix-vector operation to calculate error bounds.
?la_gbrcond	s, d	Estimates the Skeel condition number for a general banded matrix.
?la_gbrcond_c	c, z	Computes the infinity norm condition number of $op(A)*inv(diag(c))$ for general banded matrices.
?la_gbrcond_x	c, z	Computes the infinity norm condition number of $op(A)*diag(x)$ for general banded matrices.
?la_gbrfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_gbrpvgrw	s, d, c, z	Computes the reciprocal pivot growth factor $norm(A)/norm(U)$ for a general banded matrix.
?la_geamv	s, d, c, z	Computes a matrix-vector product using a general matrix to calculate error bounds.
?la_gercond	s, d	Estimates the Skeel condition number for a general matrix.
?la_gercond_c	c, z	Computes the infinity norm condition number of $op(A)*inv(diag(c))$ for general matrices.
?la_gercond_x	c, z	Computes the infinity norm condition number of $op(A)*diag(x)$ for general matrices.
?la_gerfsx_extended	s, d	Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_heamv	c, z	Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.
?la_hercond_c	c, z	Computes the infinity norm condition number of $op(A)*inv(diag(c))$ for Hermitian indefinite matrices.
?la_hercond_x	c, z	Computes the infinity norm condition number of $op(A)*diag(x)$ for Hermitian indefinite matrices.
?la_herfsx_extended	c, z	Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_lin_berr	s, d, c, z	Computes a component-wise relative backward error.
?la_porcond	s, d	Estimates the Skeel condition number for a symmetric positive-definite matrix.
?la_porcond_c	c, z	Computes the infinity norm condition number of $op(A)*inv(diag(c))$ for Hermitian positive-definite matrices.

Routine Name	Data Types	Description
<code>?la_porcond_x</code>	<i>c, z</i>	Computes the infinity norm condition number of $\text{op}(A) \cdot \text{diag}(x)$ for Hermitian positive-definite matrices.
<code>? la_porfsx_extended</code>	<i>s, d, c, z</i>	Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
<code>?la_porpvgrw</code>	<i>s, d, c, z</i>	Computes the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ for a symmetric or Hermitian positive-definite matrix.
<code>?laqhe</code>	<i>c, z</i>	Scales a Hermitian matrix.
<code>?laqhp</code>	<i>c, z</i>	Scales a Hermitian matrix stored in packed form.
<code>?larcm</code>	<i>c, z</i>	Copies all or part of a real two-dimensional array to a complex array.
<code>?la_rpvgrw</code>	<i>c, z</i>	Multiplies a square real matrix by a complex matrix.
<code>?larscl2</code>	<i>s, d, c, z</i>	Performs reciprocal diagonal scaling on a vector.
<code>?lascl2</code>	<i>s, d, c, z</i>	Performs diagonal scaling on a vector.
<code>?la_syamv</code>	<i>s, d, c, z</i>	Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.
<code>?la_syrcond</code>	<i>s, d</i>	Estimates the Skeel condition number for a symmetric indefinite matrix.
<code>?la_syrcond_c</code>	<i>c, z</i>	Computes the infinity norm condition number of $\text{op}(A) \cdot \text{inv}(\text{diag}(c))$ for symmetric indefinite matrices.
<code>?la_syrcond_x</code>	<i>c, z</i>	Computes the infinity norm condition number of $\text{op}(A) \cdot \text{diag}(x)$ for symmetric indefinite matrices.
<code>? la_syrf sx_extended</code>	<i>s, d, c, z</i>	Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
<code>?la_syrpvgrw</code>	<i>s, d, c, z</i>	Computes the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ for a symmetric indefinite matrix.
<code>?la_wwaddw</code>	<i>s, d, c, z</i>	Adds a vector into a doubled-single vector.
<code>mkl_?tppack</code>	<i>s, d, c, z</i>	Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.
<code>mkl_?tpunpack</code>	<i>s, d, c, z</i>	Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.

?lacgv*Conjugates a complex vector.***Syntax**

```
call clacgv( n, x, incx )
```

```
call zlacgv( n, x, incx )
```

Include Files

- `mkl.fi`

Description

The routine conjugates a complex vector x of length n and increment $incx$ (see "Vector Arguments in BLAS" in Appendix B).

Input Parameters

The data types are given for the Fortran interface.

n	INTEGER. The length of the vector x ($n \geq 0$).
x	COMPLEX for <code>clacgv</code> DOUBLE COMPLEX for <code>zlacgv</code> . Array, dimension $(1+(n-1) * incx)$. Contains the vector of length n to be conjugated.
$incx$	INTEGER. The spacing between successive elements of x .

Output Parameters

x	On exit, overwritten with <code>conjg(x)</code> .
-----	---

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

```
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- `mkl.fi`

Description

The routine performs a simple matrix-matrix multiplication of the form

$$C = A * B,$$

where A is m -by- n and complex, B is n -by- n and real, C is m -by- n and complex.

Input Parameters

m	INTEGER. The number of rows of the matrix A and of the matrix C ($m \geq 0$).
n	INTEGER. The number of columns and rows of the matrix B and the number of columns of the matrix C ($n \geq 0$).
a	COMPLEX for <code>clacrm</code>

	DOUBLE COMPLEX for <code>zlacrm</code>
	Array, DIMENSION(<code>lda</code> , <code>n</code>). Contains the m -by- n matrix A .
<code>lda</code>	INTEGER. The leading dimension of the array a , $lda \geq \max(1, m)$.
<code>b</code>	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code>
	Array, DIMENSION(<code>ldb</code> , <code>n</code>). Contains the n -by- n matrix B .
<code>ldb</code>	INTEGER. The leading dimension of the array b , $ldb \geq \max(1, n)$.
<code>ldc</code>	INTEGER. The leading dimension of the output array c , $ldc \geq \max(1, n)$.
<code>rwork</code>	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code>
	Workspace array, DIMENSION($2 * m * n$).

Output Parameters

<code>c</code>	COMPLEX for <code>clacrm</code> DOUBLE COMPLEX for <code>zlacrm</code>
	Array, DIMENSION(<code>ldc</code> , <code>n</code>). Contains the m -by- n matrix C .

?lacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

```
call clacrt( n, cx, incx, cy, incy, c, s )
call zlacrt( n, cx, incx, cy, incy, c, s )
```

Include Files

- `mkl.fi`

Description

The routine performs the following transformation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix},$$

where c , s are complex scalars and x , y are complex vectors.

Input Parameters

<code>n</code>	INTEGER. The number of elements in the vectors cx and cy ($n \geq 0$).
----------------	--

cx, *cy* COMPLEX for `clacrt`
 DOUBLE COMPLEX for `zlacrt`
 Arrays, dimension (*n*).
 Contain input vectors *x* and *y*, respectively.

incx INTEGER. The increment between successive elements of *cx*.

incy INTEGER. The increment between successive elements of *cy*.

c, *s* COMPLEX for `clacrt`
 DOUBLE COMPLEX for `zlacrt`
 Complex scalars that define the transform matrix



Output Parameters

cx On exit, overwritten with $c*x + s*y$.

cy On exit, overwritten with $-s*x + c*y$.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

```
call claesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
call zlaesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
```

Include Files

- `mkl.fi`

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix},$$

provided the norm of the matrix of eigenvectors is larger than some threshold value.

Input Parameters

<i>n</i>	INTEGER. The number of elements in the vectors <i>cx</i> and <i>cy</i> .
<i>cx, cy</i>	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Arrays of dimension (<i>n</i>), contain input vectors <i>x</i> and <i>y</i> , respectively.
<i>incx</i>	INTEGER. The increment between successive elements of <i>cx</i> .
<i>incy</i>	INTEGER. The increment between successive elements of <i>cy</i> .
<i>c</i>	REAL for crot DOUBLE PRECISION for zrot
<i>s</i>	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Values that define a rotation

$$\begin{bmatrix} c & s \\ -\text{conjg}(s) & c \end{bmatrix}$$

where $c*c + s*\text{conjg}(s) = 1.0$.

Output Parameters

<i>cx</i>	On exit, overwritten with $c*x + s*y$.
<i>cy</i>	On exit, overwritten with $-\text{conjg}(s)*x + c*y$.

?spm

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

```
call cspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Include Files

- mkl.fi

Description

The `?spmv` routines perform a matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

α and β are complex scalars,

x and y are n -element complex vectors

a is an n -by- n complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see `?spsmv` in Chapter 2).

Input Parameters

<code>uplo</code>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix a is supplied in the packed array ap.</p> <p>If <code>uplo = 'U' or 'u'</code>, the upper triangular part of the matrix a is supplied in the array ap.</p> <p>If <code>uplo = 'L' or 'l'</code>, the lower triangular part of the matrix a is supplied in the array ap.</p>
<code>n</code>	<p>INTEGER.</p> <p>Specifies the order of the matrix a.</p> <p>The value of n must be at least zero.</p>
<code>alpha, beta</code>	<p>COMPLEX for <code>cspmv</code></p> <p>DOUBLE COMPLEX for <code>zspmv</code></p> <p>Specify complex scalars α and β. When β is supplied as zero, then y need not be set on input.</p>
<code>ap</code>	<p>COMPLEX for <code>cspmv</code></p> <p>DOUBLE COMPLEX for <code>zspmv</code></p> <p>Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry, with <code>uplo = 'U' or 'u'</code>, the array ap must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap(1)$ contains $A(1, 1)$, $ap(2)$ and $ap(3)$ contain $A(1, 2)$ and $A(2, 2)$ respectively, and so on. Before entry, with <code>uplo = 'L' or 'l'</code>, the array ap must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1, 1)$, $ap(2)$ and $ap(3)$ contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.</p>
<code>x</code>	<p>COMPLEX for <code>cspmv</code></p> <p>DOUBLE COMPLEX for <code>zspmv</code></p> <p>Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n-element vector x.</p>
<code>incx</code>	<p>INTEGER. Specifies the increment for the elements of x. The value of <code>incx</code> must not be zero.</p>
<code>y</code>	<p>COMPLEX for <code>cspmv</code></p>

DOUBLE COMPLEX for `zsprmv`

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array `y` must contain the n -element vector `y`.

`incy` INTEGER. Specifies the increment for the elements of `y`. The value of `incy` must not be zero.

Output Parameters

`y` Overwritten by the updated vector `y`.

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
```

```
call zspr( uplo, n, alpha, x, incx, ap )
```

Include Files

- `mkl.fi`

Description

The `?spr` routines perform a matrix-vector operation defined as

$$a := \alpha * x * x^H + a,$$

where:

`alpha` is a complex scalar

`x` is an n -element complex vector

`a` is an n -by- n complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see `?spr` in Chapter 2).

Input Parameters

`uplo` CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix `a` is supplied in the packed array `ap`, as follows:

If `uplo = 'U'` or `'u'`, the upper triangular part of the matrix `a` is supplied in the array `ap`.

If `uplo = 'L'` or `'l'`, the lower triangular part of the matrix `a` is supplied in the array `ap`.

`n` INTEGER.

Specifies the order of the matrix `a`.

The value of `n` must be at least zero.

`alpha` COMPLEX for `cspr`

	DOUBLE COMPLEX for <code>zspr</code> Specifies the scalar <i>alpha</i> .
<code>x</code>	COMPLEX for <code>cspr</code> DOUBLE COMPLEX for <code>zspr</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <code>x</code> must contain the <i>n</i> -element vector <code>x</code> .
<code>incx</code>	INTEGER. Specifies the increment for the elements of <code>x</code> . The value of <code>incx</code> must not be zero.
<code>ap</code>	COMPLEX for <code>cspr</code> DOUBLE COMPLEX for <code>zspr</code> Array, DIMENSION at least $((n * (n + 1)) / 2)$. Before entry, with <code>uplo = 'U'</code> or <code>'u'</code> , the array <code>ap</code> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <code>ap(1)</code> contains $A(1,1)$, <code>ap(2)</code> and <code>ap(3)</code> contain $A(1,2)$ and $A(2,2)$ respectively, and so on. Before entry, with <code>uplo = 'L'</code> or <code>'l'</code> , the array <code>ap</code> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <code>ap(1)</code> contains $a(1,1)$, <code>ap(2)</code> and <code>ap(3)</code> contain $a(2,1)$ and $a(3,1)$ respectively, and so on. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

<code>ap</code>	With <code>uplo = 'U'</code> or <code>'u'</code> , overwritten by the upper triangular part of the updated matrix. With <code>uplo = 'L'</code> or <code>'l'</code> , overwritten by the lower triangular part of the updated matrix.
-----------------	--

?syconv

Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.

Syntax

```
call ssyconv( uplo, way, n, a, lda, ipiv, work, info )
call dsyconv( uplo, way, n, a, lda, ipiv, work, info )
call csyconv( uplo, way, n, a, lda, ipiv, work, info )
call zsyconv( uplo, way, n, a, lda, ipiv, work, info )
call syconv( a[,uplo][,way][,ipiv][,info] )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine converts matrix A , which results from a triangular matrix factorization, into matrices L and D and vice versa. The routine gets non-diagonalized elements of D returned in the workspace and applies or reverses permutation done with the triangular matrix factorization.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the details of the factorization are stored as an upper or lower triangular matrix: If <i>uplo</i> = 'U': the upper triangular, $A = U*D*U^T$. If <i>uplo</i> = 'L': the lower triangular, $A = L*D*L^T$.
<i>way</i>	CHARACTER*1. Must be 'C' or 'R'.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for <i>ssyconv</i> DOUBLE PRECISION for <i>dsyconv</i> COMPLEX for <i>csyconv</i> DOUBLE COMPLEX for <i>zsyconv</i> Array of size <i>lda</i> by <i>n</i> . The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <i>?sytrf</i> .
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Details of the interchanges and the block structure of D , as returned by <i>?sytrf</i> .
<i>work</i>	INTEGER. Workspace array, size at least $\max(1, n)$.

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = -1011, memory allocation error occurred.
-------------	--

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *syconv* interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'.

<i>way</i>	Must be 'C' or 'R'.
<i>ipiv</i>	Holds the vector of length n .

See Also

[?sytrf](#)

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Include Files

- mkl.fi

Description

The routine performs the matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

alpha and *beta* are complex scalars

x and *y* are n -element complex vectors

a is an n -by- n symmetric complex matrix.

These routines have their real equivalents in BLAS (see [?symv](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha, beta</i>	COMPLEX for <i>csymv</i> DOUBLE COMPLEX for <i>zsymv</i> Specify the scalars <i>alpha</i> and <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>a</i>	COMPLEX for <i>csymv</i> DOUBLE COMPLEX for <i>zsymv</i>

Array, DIMENSION (*lda*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced. Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. Specifies the leading dimension of *A* as declared in the calling (sub)program. The value of *lda* must be at least $\max(1, n)$.

x COMPLEX for *csymv*
 DOUBLE COMPLEX for *zsymv*
 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array *x* must contain the *n*-element vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*. The value of *incx* must not be zero.

y COMPLEX for *csymv*
 DOUBLE COMPLEX for *zsymv*
 Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array *y* must contain the *n*-element vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*. The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
call csyr( uplo, n, alpha, x, incx, a, lda )
```

```
call zsym( uplo, n, alpha, x, incx, a, lda )
```

Include Files

- `mk1.fi`

Description

The routine performs the symmetric rank 1 operation defined as

$$a := \alpha * x * x^H + a,$$

where:

- *alpha* is a complex scalar.

- x is an n -element complex vector.
- a is an n -by- n complex symmetric matrix.

These routines have their real equivalents in BLAS (see [?syr](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array a is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array a is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array a is used.
<i>n</i>	INTEGER. Specifies the order of the matrix a . The value of n must be at least zero.
<i>alpha</i>	COMPLEX for <i>csyr</i> DOUBLE COMPLEX for <i>zsyr</i> Specifies the scalar α .
<i>x</i>	COMPLEX for <i>csyr</i> DOUBLE COMPLEX for <i>zsyr</i> Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element vector x .
<i>incx</i>	INTEGER. Specifies the increment for the elements of x . The value of <i>incx</i> must not be zero.
<i>a</i>	COMPLEX for <i>csyr</i> DOUBLE COMPLEX for <i>zsyr</i> Array, size (lda, n) . Before entry with <i>uplo</i> = 'U' or 'u', the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.
<i>lda</i>	INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the i -th parameter had an illegal value. If <i>info</i> = -1011, memory allocation error occurred.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

```
index = icmax1( n, cx, incx )
```

```
index = izmax1( n, cx, incx )
```

Include Files

- mkl.fi

Description

Given a complex vector *cx*, the *i?max1* functions return the index of the first vector element of maximum absolute value. These functions are based on the BLAS functions *icamax/izamax*, but using the absolute value of components. They are designed for use with *clacon/zlacon*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in the vector <i>cx</i> .
<i>cx</i>	COMPLEX for <i>icmax1</i> DOUBLE COMPLEX for <i>izmax1</i> Array, size at least $(1+(n-1)*abs(incx))$. Contains the input vector.
<i>incx</i>	INTEGER. Specifies the spacing between successive elements of <i>cx</i> .

Output Parameters

<i>index</i>	INTEGER. Index of the vector element of maximum absolute value.
--------------	---

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

```
res = ssum1( n, cx, incx )
```

```
res = dsum1( n, cx, incx )
```

Include Files

- mkl.fi

Description

Given a complex vector *cx*, *ssum1/dsum1* functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on *scasum/dzasum* from Level 1 BLAS, but use the true absolute value and were designed for use with *clacon/zlacon*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in the vector <i>cx</i> .
<i>cx</i>	COMPLEX for <i>sccsum1</i> DOUBLE COMPLEX for <i>dzsum1</i> Array, size at least $(1+(n-1)*abs(incx))$. Contains the input vector whose elements will be summed.
<i>incx</i>	INTEGER. Specifies the spacing between successive elements of <i>cx</i> (<i>incx</i> > 0).

Output Parameters

<i>res</i>	REAL for <i>sccsum1</i> DOUBLE PRECISION for <i>dzsum1</i> Sum of absolute values.
------------	--

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtf2( m, n, kl, ku, ab, ldab, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine forms the *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also [?gbtrf](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab</i>	REAL for <i>sgbtf2</i>

DOUBLE PRECISION for `dgbtf2`

COMPLEX for `cgbtf2`

DOUBLE COMPLEX for `zgbtf2`.

Array, DIMENSION (*ldab*,*).

The array *ab* contains the matrix *A* in band storage (see [Matrix Arguments](#)).

The second dimension of *ab* must be at least $\max(1, n)$.

ldab

INTEGER. The leading dimension of the array *ab*.

(*ldab* ≥ 2*kl* + *ku* + 1)

Output Parameters

ab

Overwritten by details of the factorization. The diagonal and *kl* + *ku* super-diagonals of *U* are stored in the first 1 + *kl* + *ku* rows of *ab*. The multipliers used during the factorization are stored in the next *kl* rows.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, \min(m, n))$.

The pivot indices: row *i* was interchanged with row *ipiv*(*i*).

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, *u*_{*ij*} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

```
call dgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

```
call cgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

```
call zgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

Include Files

- `mkl.fi`

Description

The routine reduces a general *m*-by-*n* matrix *A* to upper or lower bidiagonal form *B* by an orthogonal (unitary) transformation: $Q^T A P = B$ (for real flavors) or $Q^H A P = B$ (for complex flavors).

If $m \geq n$, *B* is upper bidiagonal; if $m < n$, *B* is lower bidiagonal.

The routine does not form the matrices *Q* and *P* explicitly, but represents them as products of elementary reflectors. if $m \geq n$,

$Q = H(1) * H(2) * \dots * H(n)$, and $P = G(1) * G(2) * \dots * G(n-1)$

if $m < n$,

$Q = H(1) * H(2) * \dots * H(m-1)$, and $P = G(1) * G(2) * \dots * G(m)$

Each $H(i)$ and $G(i)$ has the form

$H(i) = I - \tau_{uq} * v * v^T$ and $G(i) = I - \tau_{up} * u * u^T$ for real flavors, or

$H(i) = I - \tau_{uq} * v * v^H$ and $G(i) = I - \tau_{up} * u * u^H$ for complex flavors

where τ_{uq} and τ_{up} are scalars (real for *sgebd2/dgebd2*, complex for *cgebd2/zgebd2*), and v and u are vectors (real for *sgebd2/dgebd2*, complex for *cgebd2/zgebd2*).

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, *work* REAL for *sgebd2*
DOUBLE PRECISION for *dgebd2*
COMPLEX for *cgebd2*
DOUBLE COMPLEX for *zgebd2*.

Arrays:

a(*lda*,*) contains the m -by- n general matrix A to be reduced. The second dimension of *a* must be at least $\max(1, n)$.

work(*) is a workspace array, the dimension of *work* must be at least $\max(1, m, n)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

Output Parameters

a if $m \geq n$, the diagonal and first super-diagonal of *a* are overwritten with the upper bidiagonal matrix B . Elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the first superdiagonal, with the array *taup*, represent the orthogonal/unitary matrix p as a product of elementary reflectors.

if $m < n$, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix B . Elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the diagonal, with the array *taup*, represent the orthogonal/unitary matrix p as a product of elementary reflectors.

d REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$.

Contains the diagonal elements of the bidiagonal matrix B : $d(i) = a(i, i)$.

<code>e</code>	<p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n) - 1)$.</p> <p>Contains the off-diagonal elements of the bidiagonal matrix B:</p> <p>if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$;</p> <p>if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$.</p>
<code>tauq, taup</code>	<p>REAL for sgebd2</p> <p>DOUBLE PRECISION for dgebd2</p> <p>COMPLEX for cgebd2</p> <p>DOUBLE COMPLEX for zgebd2.</p> <p>Arrays, DIMENSION at least $\max(1, \min(m, n))$.</p> <p>Contain scalar factors of the elementary reflectors which represent orthogonal/unitary matrices Q and p, respectively.</p>
<code>info</code>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the ith parameter had an illegal value.</p>

?gehd2

Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.

Syntax

```
call sgehd2( n, ilo, ihi, a, lda, tau, work, info )
call dgehd2( n, ilo, ihi, a, lda, tau, work, info )
call cgehd2( n, ilo, ihi, a, lda, tau, work, info )
call zgehd2( n, ilo, ihi, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine reduces a real/complex general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $Q^T A Q = H$ (for real flavors) or $Q^{H*} A Q = H$ (for complex flavors).

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*.

Input Parameters

<code>n</code>	INTEGER The order of the matrix A ($n \geq 0$).
<code>ilo, ihi</code>	INTEGER. It is assumed that A is already upper triangular in rows and columns $1:ilo - 1$ and $ihi+1:n$.

If A has been output by `?gebal`, then

ilo and ihi must contain the values returned by that routine. Otherwise they should be set to $ilo = 1$ and $ihi = n$. Constraint: $1 \leq ilo \leq ihi \leq \max(1, n)$.

$a, work$ REAL for `sgehd2`
DOUBLE PRECISION for `dgehd2`
COMPLEX for `cgehd2`
DOUBLE COMPLEX for `zgehd2`.

Arrays:

a ($lda, *$) contains the n -by- n matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$.

$work$ (n) is a workspace array.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

a On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H and the elements below the first subdiagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors. See *Application Notes* below.

tau REAL for `sgehd2`
DOUBLE PRECISION for `dgehd2`
COMPLEX for `cgehd2`
DOUBLE COMPLEX for `zgehd2`.
Array, DIMENSION at least $\max(1, n-1)$.
Contains the scalar factors of elementary reflectors. See *Application Notes* below.

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The matrix Q is represented as a product of $(ihi - ilo)$ elementary reflectors

$$Q = H(ilo) * H(ilo + 1) * \dots * H(ihi - 1)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - tau * v * v^H \text{ for complex flavors}$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(ihi + 1:n) = 0$.

On exit, $v(i+2:ihi)$ is stored in $a(i+2:ihi, i)$ and tau in $tau(i)$.

The contents of a are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & a & a & a & a & a \\ & & & a & a & a & a \\ & & & & a & a & a \\ & & & & & a & a \\ & & & & & & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & & h & h & h & h & h \\ & & & v_2 & h & h & h \\ & & & & v_2 & v_3 & h \\ & & & & & v_2 & v_3 & v_4 & h & h & h \\ & & & & & & & & & & a \end{bmatrix}$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

sgelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2( m, n, a, lda, tau, work, info )
call dgelq2( m, n, a, lda, tau, work, info )
call cgelq2( m, n, a, lda, tau, work, info )
call zgelq2( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes an LQ factorization of a real/complex m -by- n matrix A as $A = L^*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$$Q = H(k) \dots H(2) H(1) \text{ (or } Q = H(k)^H \dots H(2)^H H(1)^H \text{ for complex flavors), where } k = \min(m, n)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau v v^H \text{ for complex flavors,}$$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:n}$ (for real functions) and $\text{conjg}(v_{i+1:n})$ (for complex functions) are stored in $a(i, i+1:n)$.

Input Parameters

The data types are given for the Fortran interface.

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for <i>sgelq2</i> DOUBLE PRECISION for <i>dgelq2</i> COMPLEX for <i>cgelq2</i> DOUBLE COMPLEX for <i>zgelq2</i> . Arrays: <i>a(lda,*)</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work(m)</i> is a workspace array.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: on exit, the elements on and below the diagonal of the array <i>a</i> contain the <i>m</i> -by- $\min(n, m)$ lower trapezoidal matrix <i>L</i> (<i>L</i> is lower triangular if $n \geq m$); the elements above the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of $\min(n, m)$ elementary reflectors.
<i>tau</i>	REAL for <i>sgelq2</i> DOUBLE PRECISION for <i>dgelq2</i> COMPLEX for <i>cgelq2</i> DOUBLE COMPLEX for <i>zgelq2</i> . Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = -1011, memory allocation error occurred.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2( m, n, a, lda, tau, work, info )
call dgeql2( m, n, a, lda, tau, work, info )
call cgeql2( m, n, a, lda, tau, work, info )
```

call `zgeql2(m, n, a, lda, tau, work, info)`

Include Files

- `mkl.fi`

Description

The routine computes a *QL* factorization of a real/complex *m*-by-*n* matrix *A* as $A = Q * L$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors* :

$$Q = H(k) * \dots * H(2) * H(1), \text{ where } k = \min(m, n).$$

Each *H*(*i*) has the form

$$H(i) = I - \tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau * v * v^H \text{ for complex flavors}$$

where *tau* is a real/complex scalar stored in *tau*(*i*), and *v* is a real/complex vector with $v_{(m-k+i+1:m)} = 0$ and $v_{(m-k+i)} = 1$.

On exit, $v_{(1:m-k+i-1)}$ is stored in $a_{(1:m-k+i-1, n-k+i)}$.

Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for <code>sgeql2</code> DOUBLE PRECISION for <code>dgeql2</code> COMPLEX for <code>cgeql2</code> DOUBLE COMPLEX for <code>zgeql2</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (<i>m</i>) is a workspace array.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: on exit, if $m \geq n$, the lower triangle of the subarray $a_{(m-n+1:m, 1:n)}$ contains the <i>n</i> -by- <i>n</i> lower triangular matrix <i>L</i> ; if $m < n$, the elements on and below the (<i>n</i> - <i>m</i>)th superdiagonal contain the <i>m</i> -by- <i>n</i> lower trapezoidal matrix <i>L</i> ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors.
<i>tau</i>	REAL for <code>sgeql2</code> DOUBLE PRECISION for <code>dgeql2</code> COMPLEX for <code>cgeql2</code>

DOUBLE COMPLEX for `zgeql2`.

Array, DIMENSION at least $\max(1, \min(m, n))$.

Contains scalar factors of the elementary reflectors.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2( m, n, a, lda, tau, work, info )
```

```
call dgeqr2( m, n, a, lda, tau, work, info )
```

```
call cgeqr2( m, n, a, lda, tau, work, info )
```

```
call zgeqr2( m, n, a, lda, tau, work, info )
```

Include Files

- `mkl.fi`

Description

The routine computes a QR factorization of a real/complex *m*-by-*n* matrix *A* as $A = Q^*R$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(1) * H(2) * \dots * H(k)$, where $k = \min(m, n)$

Each *H*(*i*) has the form

$H(i) = I - \tau v v^T$ for real flavors, or

$H(i) = I - \tau v v^H$ for complex flavors

where *tau* is a real/complex scalar stored in *tau*(*i*), and *v* is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:m}$ is stored in *a*(*i*+1:*m*, *i*).

Input Parameters

The data types are given for the Fortran interface.

m INTEGER. The number of rows in the matrix *A* ($m \geq 0$).

n INTEGER. The number of columns in *A* ($n \geq 0$).

a, *work* REAL for `sgeqr2`

DOUBLE PRECISION for `dgeqr2`

COMPLEX for `cgeqr2`

DOUBLE COMPLEX for `zgeqr2`.

Arrays:

`a(lda,*)` contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

`work(n)` is a workspace array.

`lda` INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

`a` Overwritten by the factorization data as follows:
on exit, the elements on and above the diagonal of the array a contain the $\min(n,m)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

`tau` REAL for `sgeqr2`
DOUBLE PRECISION for `dgeqr2`
COMPLEX for `cgeqr2`
DOUBLE COMPLEX for `zgeqr2`.
Array, size at least $\max(1, \min(m, n))$.
Contains scalar factors of the elementary reflectors.

`info` INTEGER.
If `info = 0`, the execution is successful.
If `info = -i`, the i -th parameter had an illegal value.
If `info = -1011`, memory allocation error occurred.

?geqr2p

Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.

Syntax

```
call sgeqr2p( m, n, a, lda, tau, work, info )
call dgeqr2p( m, n, a, lda, tau, work, info )
call cgeqr2p( m, n, a, lda, tau, work, info )
call zgeqr2p( m, n, a, lda, tau, work, info )
```

Include Files

- `mkl.fi`

Description

The routine computes a QR factorization of a real/complex m -by- n matrix A as $A = Q^*R$. The diagonal entries of R are real and nonnegative.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$$Q = H(1) * H(2) * \dots * H(k), \text{ where } k = \min(m, n)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau * v * v^H \text{ for complex flavors}$$

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:m)$ is stored in $a(i+1:m, i)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for `sgeqr2p`
 DOUBLE PRECISION for `d`
 COMPLEX for `cgeqr2p`
 DOUBLE COMPLEX for `zgeqr2p`.

Arrays:
a(*lda*,*) contains the m -by- n matrix A .
 The second dimension of *a* must be at least $\max(1, n)$.
work(*n*) is a workspace array.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

Output Parameters

a Overwritten by the factorization data as follows:
 on exit, the elements on and above the diagonal of the array *a* contain the $\min(n,m)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$).
 The diagonal entries of R are real and nonnegative; the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

tau REAL for `sgeqr2p`
 DOUBLE PRECISION for `dgeqr2p`
 COMPLEX for `cgeqr2p`
 DOUBLE COMPLEX for `zgeqr2p`.

Array, DIMENSION at least $\max(1, \min(m, n))$.
 Contains scalar factors of the elementary reflectors.

info INTEGER.
 If *info* = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

zgeqrt2

Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q .

Syntax

```
call sgeqrt2(m, n, a, lda, t, ldt, info)
call dgeqrt2(m, n, a, lda, t, ldt, info)
call cgeqrt2(m, n, a, lda, t, ldt, info)
call zgeqrt2(m, n, a, lda, t, ldt, info)
call ggeqrt2(a, t, [info])
```

Include Files

- mkl.fi, lapack.f90

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

where v_i represents the vector that defines $H(i)$. The vectors are returned in the lower triangular part of array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$H = I - V^* T^* V^T$ for real flavors, and

$H = I - V^* T^* V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq n$).

<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i>	REAL for <i>sgeqrt2</i> DOUBLE PRECISION for <i>dgeqrt2</i> COMPLEX for <i>cgeqrt2</i> COMPLEX*16 for <i>zgeqrt2</i> . Array, size <i>lda</i> by <i>n</i> . Array <i>a</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldt</i>	INTEGER. The leading dimension of <i>t</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the <i>n</i> -by- <i>n</i> upper triangular matrix <i>R</i> . The elements below the diagonal are the columns of <i>V</i> .
<i>t</i>	REAL for <i>sgeqrt2</i> DOUBLE PRECISION for <i>dgeqrt2</i> COMPLEX for <i>cgeqrt2</i> COMPLEX*16 for <i>zgeqrt2</i> . Array, size (<i>ldt</i> , $\min(m, n)$). The <i>n</i> -by- <i>n</i> upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector <i>T</i> . The elements below the diagonal are not used.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0 and <i>info</i> = - <i>i</i> , the <i>i</i> th argument had an illegal value. If <i>info</i> = -1011, memory allocation error occurred.

?geqrt3

Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
call sgeqrt3(m, n, a, lda, t, ldt, info)
call dgeqrt3(m, n, a, lda, t, ldt, info)
call cgeqrt3(m, n, a, lda, t, ldt, info)
call zgeqrt3(m, n, a, lda, t, ldt, info)
call geqrt3(a, t [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower part of triangular array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$$H = I - V^* T^* V^T \text{ for real flavors, and}$$

$$H = I - V^* T^* V^H \text{ for complex flavors,}$$

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq n$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a	REAL for <code>sgeqrt3</code> DOUBLE PRECISION for <code>dgeqrt3</code> COMPLEX for <code>cgeqrt3</code> COMPLEX*16 for <code>zgeqrt3</code> . Array, size (lda, n) . Array a contains the m -by- n matrix A .
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The elements on and above the diagonal of the array contain the n -by- n upper triangular matrix R . The elements below the diagonal are the columns of V .
<i>t</i>	REAL for sgeqrt3 DOUBLE PRECISION for dgeqrt3 COMPLEX for cgeqrt3 COMPLEX*16 for zgeqrt3. Array, size ldt by n . The n -by- n upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T . The elements below the diagonal are not used.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info < 0$ and $info = -i$, the i th argument had an illegal value. If $info = -1011$, memory allocation error occurred.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2( m, n, a, lda, tau, work, info )
call dgerq2( m, n, a, lda, tau, work, info )
call cgerq2( m, n, a, lda, tau, work, info )
call zgerq2( m, n, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine computes a RQ factorization of a real/complex m -by- n matrix A as $A = R^*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or

$Q = H(1)^H * H(2)^H * \dots * H(k)^H$ for complex flavors

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v^T$ for real flavors, or

$H(i) = I - \tau * v * v^H$ for complex flavors

where τ is a real/complex scalar stored in $\tau(i)$, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$.

On exit, $v(1:n-k+i-1)$ is stored in $a(m-k+i, 1:n-k+i-1)$.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for sgerq2 DOUBLE PRECISION for dgerq2 COMPLEX for cgerq2 DOUBLE COMPLEX for zgerq2. Arrays: $a(lda, *)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $work(m)$ is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

a	Overwritten by the factorization data as follows: on exit, if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix R ; if $m > n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
τ	REAL for sgerq2 DOUBLE PRECISION for dgerq2 COMPLEX for cgerq2 DOUBLE COMPLEX for zgerq2. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.

Syntax

```
call sgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call dgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call cgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
call zgesc2( n, a, lda, rhs, ipiv, jpiv, scale )
```

Include Files

- mkl.fi

Description

The routine solves a system of linear equations

$$A * X = scale * RHS$$

with a general n -by- n matrix A using the LU factorization with complete pivoting computed by [?getc2](#).

Input Parameters

n INTEGER. The order of the matrix A .

a, rhs REAL for sgesc2
DOUBLE PRECISION for dgesc2
COMPLEX for cgesc2
DOUBLE COMPLEX for zgesc2.

Arrays:
a(*lda*,*) contains the LU part of the factorization of the n -by- n matrix A computed by [?getc2](#):
 $A = P * L * U * Q$.

The second dimension of *a* must be at least $\max(1, n)$;
rhs(n) contains on entry the right hand side vector for the system of equations.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, n)$.
The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row *ipiv*(i).

jpiv INTEGER.
Array, DIMENSION at least $\max(1, n)$.
The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column *jpiv*(j).

Output Parameters

rhs On exit, overwritten with the solution vector X .

scale REAL for sgesc2/cgesc2
 DOUBLE PRECISION for dgesc2/zgesc2

Contains the scale factor. *scale* is chosen in the range $0 \leq scale \leq 1$ to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n -by- n matrix.

Syntax

```
call sgetc2( n, a, lda, ipiv, jpiv, info )
call dgetc2( n, a, lda, ipiv, jpiv, info )
call cgetc2( n, a, lda, ipiv, jpiv, info )
call zgetc2( n, a, lda, ipiv, jpiv, info )
```

Include Files

- mkl.fi

Description

The routine computes an LU factorization with complete pivoting of the n -by- n matrix A . The factorization has the form $A = P * L * U * Q$, where P and Q are permutation matrices, L is lower triangular with unit diagonal elements and U is upper triangular.

The LU factorization computed by this routine is used by ?latdf to compute a contribution to the reciprocal Dif-estimate.

Input Parameters

n INTEGER. The order of the matrix A ($n \geq 0$).

a REAL for sgetc2
 DOUBLE PRECISION for dgetc2
 COMPLEX for cgetc2
 DOUBLE COMPLEX for zgetc2.

Array $a(lda, *)$ contains the n -by- n matrix A to be factored. The second dimension of a must be at least $\max(1, n)$;

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

a On exit, the factors L and U from the factorization $A = P * L * U * Q$; the unit diagonal elements of L are not stored. If $U(k, k)$ appears to be less than $smin$, $U(k, k)$ is given the value of $smin$, that is giving a nonsingular perturbed system.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$.

The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row $ipiv(i)$.

jpiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column $jpiv(j)$.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = k > 0$, $U(k, k)$ is likely to produce overflow if we try to solve for x in $A*x = b$. So U is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2( m, n, a, lda, ipiv, info )
```

```
call dgetf2( m, n, a, lda, ipiv, info )
```

```
call cgetf2( m, n, a, lda, ipiv, info )
```

```
call zgetf2( m, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the LU factorization of a general m -by- n matrix A using partial pivoting with row interchanges. The factorization has the form

$$A = P * L * U$$

where p is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Input Parameters

The data types are given for the Fortran interface.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for sgetf2
DOUBLE PRECISION for dgetf2
COMPLEX for cgetf2

DOUBLE COMPLEX for `zgetf2`.

Array, size $(lda,*)$. Contains the matrix A to be factored. The second dimension of a must be at least $\max(1, n)$.

`lda` INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

`a` Overwritten by L and U . The unit diagonal elements of L are not stored.

`ipiv` INTEGER.

Array, size at least $\max(1, \min(m, n))$.

The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv(i)$.

`info` INTEGER. If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i > 0`, u_{ij} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If `info = -1011`, memory allocation error occurred.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

`gttrf`.

Syntax

`call sgtts2(itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)`

`call dgtts2(itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)`

`call cgtts2(itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)`

`call zgtts2(itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb)`

Include Files

- `mkl.fi`

Description

The routine solves for X one of the following systems of linear equations with multiple right hand sides:

$A^*X = B$, $A^T*X = B$, or $A^H*X = B$ (for complex matrices only), with a tridiagonal matrix A using the LU factorization computed by `?gttrf`.

Input Parameters

`itrans` INTEGER. Must be 0, 1, or 2.

Indicates the form of the equations to be solved:

If `itrans = 0`, then $A^*X = B$ (no transpose).

	If $itrans = 1$, then $A^T * X = B$ (transpose).
	If $itrans = 2$, then $A^H * X = B$ (conjugate transpose).
n	INTEGER. The order of the matrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right-hand sides, i.e., the number of columns in B ($nrhs \geq 0$).
$dl, d, du, du2, b$	REAL for <code>sgtts2</code> DOUBLE PRECISION for <code>dgtts2</code> COMPLEX for <code>cgtts2</code> DOUBLE COMPLEX for <code>zgtts2</code> . Arrays: $dl(n - 1), d(n), du(n - 1), du2(n - 2), b(l, n)$. The array dl contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A . The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array du contains the $(n - 1)$ elements of the first super-diagonal of U . The array $du2$ contains the $(n - 2)$ elements of the second super-diagonal of U . The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
ldb	INTEGER. The leading dimension of b ; must be $ldb \geq \max(1, n)$.
$ipiv$	INTEGER. Array, DIMENSION (n) . The pivot indices array, as returned by <code>?gttrf</code> .

Output Parameters

b Overwritten by the solution matrix X .

?isnan

Tests input for NaN.

Syntax

$val = \text{sisnan}(sin)$

$val = \text{disnan}(din)$

Include Files

- `mkl.fi`

Description

This logical routine returns `.TRUE.` if its argument is NaN, and `.FALSE.` otherwise.

Input Parameters

<i>sin</i>	REAL for <i>sisnan</i> Input to test for NaN.
<i>din</i>	DOUBLE PRECISION for <i>disnan</i> Input to test for NaN.

Output Parameters

<i>val</i>	Logical. Result of the test.
------------	------------------------------

?laisnan

Tests input for NaN.

Syntax

```
val = slaisnan( sin1, sin2 )
val = dlaisnan( din1, din2 )
```

Include Files

- `mkl.fi`

Description

This logical routine checks for NaNs (NaN stands for 'Not A Number') by comparing its two arguments for inequality. NaN is the only floating-point value where `NaN ≠ NaN` returns `.TRUE`. To check for NaNs, pass the same variable as both arguments.

This routine is not for general use. It exists solely to avoid over-optimization in `?isnan`.

Input Parameters

<i>sin1, sin2</i>	REAL for <i>sisnan</i> Two numbers to compare for inequality.
<i>din2, din2</i>	DOUBLE PRECISION for <i>disnan</i> Two numbers to compare for inequality.

Output Parameters

<i>val</i>	Logical. Result of the comparison.
------------	------------------------------------

?labrd

Reduces the first nb rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

```
call clabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call zlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Include Files

- mkl.fi

Description

The routine reduces the first nb rows and columns of a general m -by- n matrix A to upper or lower bidiagonal form by an orthogonal/unitary transformation Q^*A^*P , and returns the matrices X and Y which are needed to apply the transformation to the unreduced part of A .

if $m \geq n$, A is reduced to upper bidiagonal form; if $m < n$, to lower bidiagonal form.

The matrices Q and P are represented as products of elementary reflectors: $Q = H(1)^*(2)^* \dots *H(nb)$, and $P = G(1)^*G(2)^* \dots *G(nb)$

Each $H(i)$ and $G(i)$ has the form

$$H(i) = I - \tau u v^* \text{ and } G(i) = I - \tau u^* u$$

where τu and τu^* are scalars, and v and u are vectors.

The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are needed, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $A := A - V^*Y' - X^*U'$.

This is an auxiliary routine called by `?gebrd`.

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
nb	INTEGER. The number of leading rows and columns of A to be reduced.
a	REAL for slabrd DOUBLE PRECISION for dlabrd COMPLEX for clabrd DOUBLE COMPLEX for zlabrd. Array $a(lda,*)$ contains the matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldx	INTEGER. The leading dimension of the output array x ; must be at least $\max(1, m)$.
ldy	INTEGER. The leading dimension of the output array y ; must be at least $\max(1, n)$.

Output Parameters

a	On exit, the first nb rows and columns of the matrix are overwritten; the rest of the array is unchanged.
-----	---

if $m \geq n$, elements on and below the diagonal in the first nb columns, with the array $tauq$, represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array $taup$, represent the orthogonal/unitary matrix p as a product of elementary reflectors.

if $m < n$, elements below the diagonal in the first nb columns, with the array $tauq$, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first nb rows, with the array $taup$, represent the orthogonal/unitary matrix p as a product of elementary reflectors.

d, e

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (nb) each. The array d contains the diagonal elements of the first nb rows and columns of the reduced matrix:

$$d(i) = a(i, i).$$

The array e contains the off-diagonal elements of the first nb rows and columns of the reduced matrix.

$tauq, taup$

REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Arrays, DIMENSION (nb) each. Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices Q and P , respectively.

x, y

REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Arrays, dimension $x(ldx, nb)$, $y(ldy, nb)$.

The array x contains the m -by- nb matrix X required to update the unreduced part of A .

The array y contains the n -by- nb matrix Y required to update the unreduced part of A .

Application Notes

if $m \geq n$, then for the elementary reflectors $H(i)$ and $G(i)$,

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in $a(i:m, i)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $a(i, i+1:n)$;

$tauq$ is stored in $tauq(i)$ and $taup$ in $taup(i)$.

if $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in $a(i+2:m, i)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $a(i, i+1:n)$; $tauq$ is stored in $tauq(i)$ and $taup$ in $taup(i)$.

The contents of a on exit are illustrated by the following examples with $nb = 2$:

$m = 6, n = 5 (m > n)$

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m = 5, n = 6 (m < n)$

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacn2( n, v, x, isgn, est, kase, isave )
call dlacn2( n, v, x, isgn, est, kase, isave )
call clacn2( n, v, x, est, kase, isave )
call zlacn2( n, v, x, est, kase, isave )
```

Include Files

- mkl.fi

Description

The routine estimates the 1-norm of a square, real or complex matrix A . Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n INTEGER. The order of the matrix A ($n \geq 1$).

v, x REAL for slacn2
DOUBLE PRECISION for dlacn2
COMPLEX for clacn2
DOUBLE COMPLEX for zlacn2.

Arrays, size (n) each.
 v is a workspace array.

	x is used as input after an intermediate return.
<i>isgn</i>	INTEGER. Workspace array, size (n), used with real flavors only.
<i>est</i>	REAL for <code>slacn2/clacn2</code> DOUBLE PRECISION for <code>dlacn2/zlacn2</code> On entry with <i>kase</i> set to 1 or 2, and <i>isave</i> (1) = 1, <i>est</i> must be unchanged from the previous call to the routine.
<i>kase</i>	INTEGER. On the initial call to the routine, <i>kase</i> must be set to 0.
<i>isave</i>	INTEGER. Array, size (3). Contains variables from the previous call to the routine.

Output Parameters

<i>est</i>	An estimate (a lower bound) for $\text{norm}(A)$.
<i>kase</i>	On an intermediate return, <i>kase</i> is set to 1 or 2, indicating whether x is overwritten by $A*x$ or A^T*x for real flavors and $A*x$ or A^H*x for complex flavors. On the final return, <i>kase</i> is set to 0.
<i>v</i>	On the final return, $v = A*w$, where $est = \text{norm}(v) / \text{norm}(w)$ (w is not returned).
<i>x</i>	On an intermediate return, x is overwritten by $A*x$, if <i>kase</i> = 1, A^T*x , if <i>kase</i> = 2 (for real flavors), A^H*x , if <i>kase</i> = 2 (for complex flavors), and the routine must be re-called with all the other parameters unchanged.
<i>isave</i>	This parameter is used to save variables between calls to the routine.

?lacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon( n, v, x, isgn, est, kase )
call dlacon( n, v, x, isgn, est, kase )
call clacon( n, v, x, est, kase )
call zlacon( n, v, x, est, kase )
```

Include Files

- `mkl.fi`

Description

The routine estimates the 1-norm of a square, real/complex matrix A . Reverse communication is used for evaluating matrix-vector products.

WARNING

The `?lacon` routine is not thread-safe. It is deprecated and retained for the backward compatibility only. Use the thread-safe `?lacn2` routine instead.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 1$).
v, x	REAL for <code>slacon</code> DOUBLE PRECISION for <code>dlacon</code> COMPLEX for <code>clacon</code> DOUBLE COMPLEX for <code>zlacon</code> . Arrays, DIMENSION (n) each. v is a workspace array. x is used as input after an intermediate return.
$isgn$	INTEGER. Workspace array, DIMENSION (n), used with real flavors only.
est	REAL for <code>slacon/clacon</code> DOUBLE PRECISION for <code>dlacon/zlacon</code> An estimate that with $kase=1$ or 2 should be unchanged from the previous call to <code>?lacon</code> .
$kase$	INTEGER. On the initial call to <code>?lacon</code> , $kase$ should be 0 .

Output Parameters

est	REAL for <code>slacon/clacon</code> DOUBLE PRECISION for <code>dlacon/zlacon</code> An estimate (a lower bound) for $\text{norm}(A)$.
$kase$	On an intermediate return, $kase$ will be 1 or 2 , indicating whether x should be overwritten by $A*x$ or A^T*x for real flavors and $A*x$ or A^H*x for complex flavors. On the final return from <code>?lacon</code> , $kase$ will again be 0 .
v	On the final return, $v = A*w$, where $est = \text{norm}(v)/\text{norm}(w)$ (w is not returned).

x On an intermediate return, *x* should be overwritten by
 $A*x$, if *kase* = 1,
 A^T*x , if *kase* = 2 (for real flavors),
 A^H*x , if *kase* = 2 (for complex flavors),
 and *?lacon* must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy( uplo, m, n, a, lda, b, ldb )
call dlacpy( uplo, m, n, a, lda, b, ldb )
call clacpy( uplo, m, n, a, lda, b, ldb )
call zlacpy( uplo, m, n, a, lda, b, ldb )
```

Include Files

- mkl.fi

Description

The routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix <i>A</i> to be copied to <i>B</i> . If <i>uplo</i> = 'U', the upper triangular part of <i>A</i> ; if <i>uplo</i> = 'L', the lower triangular part of <i>A</i> . Otherwise, all of the matrix <i>A</i> is copied.
<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i>	REAL for slacpy DOUBLE PRECISION for dlacpy COMPLEX for clacpy DOUBLE COMPLEX for zlacpy. Array <i>a</i> (<i>lda</i> ,*), contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. If <i>uplo</i> = 'U', only the upper triangle or trapezoid is accessed; if <i>uplo</i> = 'L', only the lower triangle or trapezoid is accessed.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

ldb INTEGER. The leading dimension of the output array *b*; $ldb \geq \max(1, m)$.

Output Parameters

b REAL for slacpy
 DOUBLE PRECISION for dlacpy
 COMPLEX for clacpy
 DOUBLE COMPLEX for zlacpy.
 Array *b*(*ldb*,*), contains the *m*-by-*n* matrix *B*.
 The second dimension of *b* must be at least $\max(1, n)$.
 On exit, $B = A$ in the locations specified by *uplo*.

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv( a, b, c, d, p, q )
call dladiv( a, b, c, d, p, q )
res = cladiv( x, y )
res = zladiv( x, y )
```

Include Files

- mkl.fi

Description

The routines sladiv/dladiv perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions cladiv/zladiv compute the result as

$res = x/y$,

where *x* and *y* are complex. The computation of *x* / *y* will not overflow on an intermediary step unless the results overflows.

The algorithm used is due to [Baudin12].

Input Parameters

a, b, c, d REAL for sladiv
 DOUBLE PRECISION for dladiv

The scalars a , b , c , and d in the above expression (for real flavors only).

x, y COMPLEX for `cladiv`
 DOUBLE COMPLEX for `zladiv`

The complex scalars x and y (for complex flavors only).

Output Parameters

p, q REAL for `sladiv`
 DOUBLE PRECISION for `dladiv`

The scalars p and q in the above expression (for real flavors only).

res COMPLEX for `cladiv`
 DOUBLE COMPLEX for `zladiv`

Contains the result of division x / y .

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

```
call sla2( a, b, c, rt1, rt2 )
call dla2( a, b, c, rt1, rt2 )
```

Include Files

- `mkl.fi`

Description

The routines `sla2/dla2` compute the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

On return, $rt1$ is the eigenvalue of larger absolute value, and $rt2$ is the eigenvalue of smaller absolute value.

Input Parameters

a, b, c REAL for `sla2`
 DOUBLE PRECISION for `dla2`

The elements a , b , and c of the 2-by-2 matrix above.

Output Parameters

$rt1, rt2$ REAL for `sla2`

DOUBLE PRECISION for `dlae2`

The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

$rt1$ is accurate to a few ulps barring over/underflow. $rt2$ may be inaccurate if there is massive cancellation in the determinant $a*c-b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute $rt2$ accurately in all cases.

Overflow is possible only if $rt1$ is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds

`underflow_threshold / macheps`.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.

Syntax

```
call slaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2,
nval, ab, c, mout, nab, work, iwork, info )
```

```
call dlaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2,
nval, ab, c, mout, nab, work, iwork, info )
```

Include Files

- `mkl.fi`

Description

The routine ?laebz contains the iteration loops which compute and use the function $n(w)$, which is the count of eigenvalues of a symmetric tridiagonal matrix T less than or equal to its argument w . It performs a choice of two types of loops:

$ijob$ =1, followed by

$ijob$ =2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are $(ab(j,1), ab(j,2)]$, $j=1, \dots, minp$. The output interval $(ab(j,1), ab(j,2)]$ will contain eigenvalues $nab(j, 1)+1, \dots, nab(j,2)$, where $1 \leq j \leq mout$.

$ijob$ =3: It performs a binary search in each input interval $(ab(j,1), ab(j,2)]$ for a point $w(j)$ such that $n(w(j)) = nval(j)$, and uses $c(j)$ as the starting point of the search. If such a $w(j)$ is found, then on output $ab(j,1) = ab(j,2) = w$. If no such $w(j)$ is found, then on output $(ab(j,1), ab(j,2)]$ will be a small interval containing the point where $n(w)$ jumps through $nval(j)$, unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form $(a, b]$, which includes b but not a .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than $overflow^{1/2} * overflow^{1/4}$ in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.

NOTE

In general, the arguments are not checked for unreasonable values.

Input Parameters

<i>ijob</i>	<p>INTEGER. Specifies what is to be done:</p> <ul style="list-style-type: none"> = 1: Compute <i>nab</i> for the initial intervals. = 2: Perform bisection iteration to find eigenvalues of <i>T</i>. = 3: Perform bisection iteration to invert $n(w)$, i.e., to find a point which has a specified number of eigenvalues of <i>T</i> to its left. Other values will cause ?laebz to return with <i>info</i>=-1.
<i>nitmax</i>	<p>INTEGER. The maximum number of "levels" of bisection to be performed, i.e., an interval of width <i>W</i> will not be made smaller than $2^{-nitmax * W}$. If not all intervals have converged after <i>nitmax</i> iterations, then <i>info</i> is set to the number of non-converged intervals.</p>
<i>n</i>	<p>INTEGER. The dimension <i>n</i> of the tridiagonal matrix <i>T</i>. It must be at least 1.</p>
<i>mmax</i>	<p>INTEGER. The maximum number of intervals. If more than <i>mmax</i> intervals are generated, then ?laebz will quit with <i>info</i>=<i>mmax</i>+1.</p>
<i>minp</i>	<p>INTEGER. The initial number of intervals. It may not be greater than <i>mmax</i>.</p>
<i>nbmin</i>	<p>INTEGER. The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.</p>
<i>abstol</i>	<p>REAL for slaebz DOUBLE PRECISION for dlaebz.</p> <p>The minimum (absolute) width of an interval. When an interval is narrower than <i>abstol</i>, or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.</p>
<i>reltol</i>	<p>REAL for slaebz DOUBLE PRECISION for dlaebz.</p> <p>The minimum relative width of an interval. When an interval is narrower than <i>abstol</i>, or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least $radix * machineepsilon$.</p>
<i>pivmin</i>	<p>REAL for slaebz DOUBLE PRECISION for dlaebz.</p>

The minimum absolute value of a "pivot" in the Sturm sequence loop. This value **must** be at least $(\max |e(j)|^{**2} |*safe_min)$ and at least *safe_min*, where *safe_min* is at least the smallest number that can divide one without overflow.

d, e, e2

REAL for slaebz

DOUBLE PRECISION for dlaebz.

Arrays, dimension (*n*) each. The array *d* contains the diagonal elements of the tridiagonal matrix *T*.

The array *e* contains the off-diagonal elements of the tridiagonal matrix *T* in positions 1 through *n*-1. *e*(*n*) is arbitrary.

The array *e2* contains the squares of the off-diagonal elements of the tridiagonal matrix *T*. *e2*(*n*) is ignored.

nval

INTEGER.

Array, dimension (*minp*).

If *ijob*=1 or 2, not referenced.

If *ijob*=3, the desired values of *n*(*w*).

ab

REAL for slaebz

DOUBLE PRECISION for dlaebz.

Array, dimension (*mmax*,2) The endpoints of the intervals. *ab*(*j*,1) is *a*(*j*), the left endpoint of the *j*-th interval, and *ab*(*j*,2) is *b*(*j*), the right endpoint of the *j*-th interval.

c

REAL for slaebz

DOUBLE PRECISION for dlaebz.

Array, dimension (*mmax*)

If *ijob*=1, ignored.

If *ijob*=2, workspace.

If *ijob*=3, then on input *c*(*j*) should be initialized to the first search point in the binary search.

nab

INTEGER.

Array, dimension (*mmax*,2)

If *ijob*=2, then on input, *nab*(*i*,*j*) should be set. It must satisfy the condition:

$n(ab(i,1)) \leq nab(i,1) \leq nab(i,2) \leq n(ab(i,2))$, which means that in interval *i* only eigenvalues $nab(i,1)+1, \dots, nab(i,2)$ are considered. Usually, $nab(i,j) = n(ab(i,j))$, from a previous call to ?laebz with *ijob*=1.

If *ijob*=3, normally, *nab* should be set to some distinctive value(s) before ?laebz is called.

work

REAL for slaebz

DOUBLE PRECISION for dlaebz.

iwork Workspace array, dimension (*mmax*).

iwork INTEGER.

Workspace array, dimension (*mmax*).

Output Parameters

nval The elements of *nval* will be reordered to correspond with the intervals in *ab*. Thus, *nval(j)* on output will not, in general be the same as *nval(j)* on input, but it will correspond with the interval $(ab(j,1), ab(j,2)]$ on output.

ab The input intervals will, in general, be modified, split, and reordered by the calculation.

mout INTEGER.

If *ijob*=1, the number of eigenvalues in the intervals.

If *ijob*=2 or 3, the number of intervals output.

If *ijob*=3, *mout* will equal *minp*.

nab If *ijob*=1, then on output *nab(i,j)* will be set to $N(ab(i,j))$.

If *ijob*=2, then on output, *nab(i,j)* will contain $\max(na(k), \min(nb(k), N(ab(i,j))))$, where *k* is the index of the input interval that the output interval $(ab(j,1), ab(j,2)]$ came from, and *na(k)* and *nb(k)* are the input values of *nab(k,1)* and *nab(k,2)*.

If *ijob*=3, then on output, *nab(i,j)* contains $N(ab(i,j))$, unless $N(w) > nval(i)$ for all search points *w*, in which case *nab(i,1)* will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see *nval* and *ab*), or unless $N(w) < nval(i)$ for all search points *w*, in which case *nab(i,2)* will not be modified.

info INTEGER.

If *info* = 0 - all intervals converged

If *info* = 1--*mmax* - the last *info* interval did not converge.

If *info* = *mmax*+1 - more than *mmax* intervals were generated

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, *?laebz* should have one or more initial intervals set up in *ab*, and *?laebz* should be called with *ijob*=1. This sets up *nab*, and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval *i* are desired, *nab(i,1)* can be increased or *nab(i,2)* decreased. For example, set *nab(i,1)*=*nab(i,2)*-1 to get the largest eigenvalue. *?laebz* is then called with *ijob*=2 and *mmax* no smaller than the value of *mout* returned by the call with *ijob*=1. After this (*ijob*=2) call, eigenvalues *nab(i,1)*+1 through *nab(i,2)* are approximately *ab(i,1)* (or *ab(i,2)*) to the tolerance specified by *abstol* and *reltol*.

(b) finding an interval (*a'*,*b'*) containing eigenvalues *w(f)*,...,*w(l)*. In this case, start with a Gershgorin interval (*a*,*b*). Set up *ab* to contain 2 search intervals, both initially (*a*,*b*). One *nval* element should contain *f*-1 and the other should contain *l*, while *c* should contain *a* and *b*, respectively. *nab(i,1)* should be -1 and

$nab(i,2)$ should be $n+1$, to flag an error if the desired interval does not lie in (a,b) . `?laebz` is then called with $ijob=3$. On exit, if $w(f-1) < w(f)$, then one of the intervals -- j -- will have $ab(j,1)=ab(j,2)$ and $nab(j,1)=nab(j,2)=f-1$, while if, to the specified tolerance, $w(f-k)=\dots=w(f+r)$, $k > 0$ and $r \geq 0$, then the interval will have $n(ab(j,1))=nab(j,1)=f-k$ and $n(ab(j,2))=nab(j,2)=f+r$. The cases $w(1) < w(1+1)$ and $w(l-r)=\dots=w(l+k)$ are handled similarly.

?laed0

Used by `?stedc`. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call slaed0( icompr, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call dlaed0( icompr, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call claed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
call zlaed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
```

Include Files

- `mk1.fi`

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors `claed0/zlaed0` compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

<i>icompr</i>	INTEGER. Used with real flavors only. If <i>icompr</i> = 0, compute eigenvalues only. If <i>icompr</i> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array <i>q</i> must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form. If <i>icompr</i> = 2, compute eigenvalues and eigenvectors of the tridiagonal matrix.
<i>qsiz</i>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <i>icompr</i> = 1).
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>d, e, rwork</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays:

$d(*)$ contains the main diagonal of the tridiagonal matrix. The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of the tridiagonal matrix. The dimension of e must be at least $\max(1, n-1)$.

$rwork(*)$ is a workspace array used in complex flavors only. The dimension of $rwork$ must be at least $(1 + 3n + 2n \log_2(n) + 3n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

$q, qstore$

REAL for slaed0

DOUBLE PRECISION for dlaed0

COMPLEX for claed0

DOUBLE COMPLEX for zlaed0.

Arrays: $q(ldq, *)$, $qstore(ldqs, *)$. The second dimension of these arrays must be at least $\max(1, n)$.

For real flavors:

If $icompq = 0$, array q is not referenced.

If $icompq = 1$, on entry, q is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.

If $icompq = 2$, on entry, q will be the identity matrix. The array $qstore$ is a workspace array referenced only when $icompq = 1$. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

For complex flavors:

On entry, q must contain an $qsiz$ -by- n matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix. The array $qstore$ is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq

INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.

$ldqs$

INTEGER. The leading dimension of the array $qstore$; $ldqs \geq \max(1, n)$.

$work$

REAL for slaed0

DOUBLE PRECISION for dlaed0.

Workspace array, used in real flavors only.

If $icompq = 0$ or 1 , the dimension of $work$ must be at least $(1 + 3n + 2n \log_2(n) + 3n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

If $icompq = 2$, the dimension of $work$ must be at least $(4n + n^2)$.

$iwork$

INTEGER.

Workspace array.

For real flavors, if $icompq = 0$ or 1 , and for complex flavors, the dimension of $iwork$ must be at least $(6 + 6n + 5n \log_2(n))$.

For real flavors, if $icompg = 2$, the dimension of $iwork$ must be at least $(3+5n)$.

Output Parameters

d	On exit, contains eigenvalues in ascending order.
e	On exit, the array is destroyed.
q	If $icompg = 2$, on exit, q contains the eigenvectors of the tridiagonal matrix.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i > 0$, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $i/(n+1)$ through $\text{mod}(i, n+1)$.

?laed1

Used by `sstedc/dstedc`. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
```

Include Files

- `mkl.fi`

Description

The routine `?laed1` computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. `?laed7` handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \text{rho} * Z * Z^T) * Q^T(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^T(\text{out})$$

where $Z = Q^T u$, u is a vector of length n with ones in the cutpnt and $(\text{cutpnt}+1)$ -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?laed2`.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine `?laed4` (as called by `?laed3`). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>d, q, work</i>	REAL for slaed1 DOUBLE PRECISION for dlaed1. Arrays: <i>d</i> (*) contains the eigenvalues of the rank-1-perturbed matrix. The dimension of <i>d</i> must be at least $\max(1, n)$. <i>q</i> (<i>ldq</i> , *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, dimension at least $(4n+n^2)$.
<i>ldq</i>	INTEGER. The leading dimension of the array <i>q</i> ; $ldq \geq \max(1, n)$.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). On entry, the permutation which separately sorts the two subproblems in <i>d</i> into ascending order.
<i>rho</i>	REAL for slaed1 DOUBLE PRECISION for dlaed1. The subdiagonal entry used to create the rank-1 modification. This parameter can be modified by ?1aed2, where it is input/output.
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq cutpnt \leq n/2$.
<i>iwork</i>	INTEGER. Workspace array, dimension $(4n)$.

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.
<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is, $d(\text{indxq}(i = 1, n))$ will be in ascending order.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed2

Used by sstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp,
coltyp, info )
```

```
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp,
coltyp, info )
```

Include Files

- mkl.fi

Description

The routine ?laed2 merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

<i>k</i>	INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation ($0 \leq k \leq n$).
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>n1</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.
<i>d, q, z</i>	REAL for slaed2 DOUBLE PRECISION for dlaed2.
	Arrays:
	<i>d</i> (*) contains the eigenvalues of the two submatrices to be combined. The dimension of <i>d</i> must be at least $\max(1, n)$.
	<i>q</i> (<i>ldq</i> , *) contains the eigenvectors of the two submatrices in the two square blocks with corners at (1,1), (n1,n1) and (n1+1,n1+1), (n,n). The second dimension of <i>q</i> must be at least $\max(1, n)$.
	<i>z</i> (*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).
<i>ldq</i>	INTEGER. The leading dimension of the array <i>q</i> ; $ldq \geq \max(1, n)$.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>).
	On entry, the permutation which separately sorts the two subproblems in <i>d</i> into ascending order. Note that elements in the second half of this permutation must first have <i>n1</i> added to their values.
<i>rho</i>	REAL for slaed2

DOUBLE PRECISION for `dlaed2`.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

indx, indxp

INTEGER.

Workspace arrays, dimension (*n*) each. Array *indx* contains the permutation used to sort the contents of *dlamda* into ascending order.

Array *indxp* contains the permutation used to place deflated values of *d* at the end of the array.

indxp(1:k) points to the nondeflated *d*-values and *indxp(k+1:n)* points to the deflated eigenvalues.

coltyp

INTEGER.

Workspace array, dimension (*n*).

During execution, a label which will indicate which of the following types a column in the *q2* matrix is:

1 : non-zero in the upper half only;

2 : dense;

3 : non-zero in the lower half only;

4 : deflated.

Output Parameters

d

On exit, *d* contains the trailing (*n-k*) updated eigenvalues (those which were deflated) sorted into increasing order.

q

On exit, *q* contains the trailing (*n-k*) updated eigenvectors (those which were deflated) in its last *n-k* columns.

z

On exit, *z* content is destroyed by the updating process.

indxq

Destroyed on exit.

rho

On exit, *rho* has been modified to the value required by `?1aed3`.

dlamda, w, q2

REAL for `s1aed2`

DOUBLE PRECISION for `d1aed2`.

Arrays: *dlamda*(*n*), *w*(*n*), *q2*($n1^2+(n-n1)^2$).

The array *dlamda* contains a copy of the first *k* eigenvalues which is used by `?1aed3` to form the secular equation.

The array *w* contains the first *k* values of the final deflation-altered *z*-vector which is passed to `?1aed3`.

The array *q2* contains a copy of the first *k* eigenvectors which is used by `?1aed3` in a matrix multiply (`sgemm/dgemm`) to solve for the new eigenvectors.

indx

INTEGER. Array, dimension (*n*).

The permutation used to arrange the columns of the deflated q matrix into three groups: the first group contains non-zero elements only at and above $n1$, the second contains non-zero elements only below $n1$, and the third is dense.

coltyp

On exit, *coltyp*(i) is the number of columns of type i , for $i=1$ to 4 only (see the definition of types in the description of *coltyp* in *Input Parameters*).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

?laed3

Used by *sstedc/dstedc*. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

```
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

Include Files

- mkl.fi

Description

The routine ?laed3 finds the roots of the secular equation, as defined by the values in d , w , and ρ , between 1 and k .

It makes the appropriate calls to ?laed4 and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the k -by- k system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

k INTEGER. The number of terms in the rational function to be solved by ?laed4 ($k \geq 0$).

n INTEGER. The number of rows and columns in the q matrix. $n \geq k$ (deflation may result in $n > k$).

$n1$ INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.

q REAL for slaed3
DOUBLE PRECISION for dlaed3.

Array $q(ldq, *)$. The second dimension of q must be at least $\max(1, n)$. Initially, the first k columns of this array are used as workspace.

ldq INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.

rho REAL for slaed3
DOUBLE PRECISION for dlaed3.

The value of the parameter in the rank one update equation. $rho \geq 0$ required.

dlambda, q2, w REAL for slaed3
DOUBLE PRECISION for dlaed3.

Arrays: $dlambda(k)$, $q2(ldq2, *)$, $w(k)$.

The first k elements of the array *dlambda* contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first k columns of the array *q2* contain the non-deflated eigenvectors for the split problem. The second dimension of *q2* must be at least $\max(1, n)$.

The first k elements of the array *w* contain the components of the deflation-adjusted updating vector.

indx INTEGER. Array, dimension (n).

The permutation used to arrange the columns of the deflated q matrix into three groups (see ?1aed2).

The rows of the eigenvectors found by ?1aed4 must be likewise permuted before the matrix multiply can take place.

ctot INTEGER. Array, dimension (4).

A count of the total number of the various types of columns in q , as described in *indx*. The fourth column type is any column which has been deflated.

s REAL for slaed3
DOUBLE PRECISION for dlaed3.

Workspace array, dimension $(n1+1)*k$.

Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

d REAL for slaed3
DOUBLE PRECISION for dlaed3.

Array, dimension at least $\max(1, n)$.

$d(i)$ contains the updated eigenvalues for $1 \leq i \leq k$.

q On exit, the columns 1 to k of q contain the updated eigenvectors.

<i>dlambda</i>	May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.
<i>w</i>	Destroyed on exit.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc. Finds a single root of the secular equation.

Syntax

```
call slaed4( n, i, d, z, delta, rho, dlam, info )
call dlaed4( n, i, d, z, delta, rho, dlam, info )
```

Include Files

- mkl.fi

Description

This routine computes the *i*-th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array *d*, and that

$$D(i) < D(j) \text{ for } i < j$$

and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(D) + \rho * Z * \text{transpose}(Z).$$

where we assume the Euclidean norm of *Z* is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

<i>n</i>	INTEGER. The length of all arrays.
<i>i</i>	INTEGER. The index of the eigenvalue to be computed; $1 \leq i \leq n$.
<i>d, z</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4 Arrays, dimension (<i>n</i>) each. The array <i>d</i> contains the original eigenvalues. It is assumed that they are in order, $d(i) < d(j)$ for $i < j$. The array <i>z</i> contains the components of the updating vector <i>Z</i> .
<i>rho</i>	REAL for slaed4 DOUBLE PRECISION for dlaed4

The scalar in the symmetric updating formula.

Output Parameters

<i>delta</i>	<p>REAL for slaed4</p> <p>DOUBLE PRECISION for dlaed4</p> <p>Array, dimension (<i>n</i>).</p> <p>If $n \neq 1$, <i>delta</i> contains $(d(j) - \lambda_i)$ in its <i>j</i>-th component. If $n = 1$, then $delta(1) = 1$. The vector <i>delta</i> contains the information necessary to construct the eigenvectors.</p>
<i>dlam</i>	<p>REAL for slaed4</p> <p>DOUBLE PRECISION for dlaed4</p> <p>The computed λ_i, the <i>i</i>-th updated eigenvalue.</p>
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = 1$, the updating process failed.</p>

?laed5

Used by sstedc/dstedc. Solves the 2-by-2 secular equation.

Syntax

```
call slaed5( i, d, z, delta, rho, dlam )
call dlaed5( i, d, z, delta, rho, dlam )
```

Include Files

- mkl.fi

Description

The routine computes the *i*-th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(D) + \rho * Z * \text{transpose}(Z)$.

The diagonal elements in the array *D* are assumed to satisfy

$$D(i) < D(j) \text{ for } i < j.$$

We also assume $\rho > 0$ and that the Euclidean norm of the vector *Z* is one.

Input Parameters

<i>i</i>	<p>INTEGER. The index of the eigenvalue to be computed;</p> <p>$1 \leq i \leq 2$.</p>
<i>d</i> , <i>z</i>	<p>REAL for slaed5</p> <p>DOUBLE PRECISION for dlaed5</p>

Arrays, dimension (2) each. The array d contains the original eigenvalues. It is assumed that $d(1) < d(2)$.

The array z contains the components of the updating vector.

ρ

REAL for slaed5

DOUBLE PRECISION for dlaed5

The scalar in the symmetric updating formula.

Output Parameters

δ

REAL for slaed5

DOUBLE PRECISION for dlaed5

Array, dimension (2).

The vector δ contains the information necessary to construct the eigenvectors.

d_{lam}

REAL for slaed5

DOUBLE PRECISION for dlaed5

The computed λ_i , the i -th updated eigenvalue.

?laed6

Used by sstedc/dstedc. Computes one Newton step in solution of the secular equation.

Syntax

```
call slaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

```
call dlaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

Include Files

- mkl.fi

Description

The routine computes the positive or negative root (closest to the origin) of

$$f(x) = \rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if $orgati = .TRUE.$ the root is between $d(2)$ and $d(3)$; otherwise it is between $d(1)$ and $d(2)$. This routine is called by ?laed4 when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

$kniter$

INTEGER.

Refer to ?laed4 for its significance.

orgati

LOGICAL.

If *orgati* = .TRUE., the needed root is between $d(2)$ and $d(3)$; otherwise it is between $d(1)$ and $d(2)$. See ?laed4 for further details.

rho

REAL for slaed6

DOUBLE PRECISION for dlaed6

Refer to the equation for $f(x)$ above.

d, z

REAL for slaed6

DOUBLE PRECISION for dlaed6

Arrays, dimension (3) each.

The array *d* satisfies $d(1) < d(2) < d(3)$.

Each of the elements in the array *z* must be positive.

finit

REAL for slaed6

DOUBLE PRECISION for dlaed6

The value of $f(x)$ at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

tau

REAL for slaed6

DOUBLE PRECISION for dlaed6

The root of the equation for $f(x)$.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, failure to converge.

?laed7

Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.

Syntax

```
call slaed7( icompq, n, qsiz, tlvl, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )
```

```
call dlaed7( icompq, n, qsiz, tlvl, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,
qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )
```

```
call claed7( n, cutpnt, qsiz, tlvl, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,
qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )
```

```
call zlaed7( n, cutpnt, qsiz, tlvl, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,
qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )
```

Include Files

- `mk1.fi`

Description

The routine `?laed7` computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, `s1aed1/d1aed1` handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \text{rho} * Z * Z^T) * Q^T(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^T(\text{out}) \text{ for real flavors, or}$$

$$T = Q(\text{in}) * (D(\text{in}) + \text{rho} * Z * Z^H) * Q^H(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^H(\text{out}) \text{ for complex flavors}$$

where $Z = Q^T * u$ for real flavors and $Z = Q^H * u$ for complex flavors, u is a vector of length n with ones in the `cutpnt` and `(cutpnt + 1)`-th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `s1aed8/d1aed8` (for real flavors) or by the routine `s1aed2/d1aed2` (for complex flavors).

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine `?laed4` (as called by `?laed9` or `?laed3`). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

<code>icompg</code>	INTEGER. Used with real flavors only. If <code>icompg = 0</code> , compute eigenvalues only. If <code>icompg = 1</code> , compute eigenvectors of original dense symmetric matrix also. On entry, the array q must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<code>n</code>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<code>cutpnt</code>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n$.
<code>qsiz</code>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <code>icompg = 1</code>).
<code>tlvls</code>	INTEGER. The total number of merging levels in the overall divide and conquer tree.
<code>curlvl</code>	INTEGER. The current level in the overall merge routine, $0 \leq \text{curlvl} \leq \text{tlvls}$.

<i>curpbm</i>	<p>INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).</p>
<i>d</i>	<p>REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension at least $\max(1, n)$. Array <i>d</i>(*) contains the eigenvalues of the rank-1-perturbed matrix.</p>
<i>q, work</i>	<p>REAL for slaed7 DOUBLE PRECISION for dlaed7 COMPLEX for claed7 DOUBLE COMPLEX for zlaed7. Arrays: <i>q</i>(<i>ldq</i>, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>work</i>(*) is a workspace array, dimension at least $(3n+2*qsiz*n)$ for real flavors and at least $(qsiz*n)$ for complex flavors.</p>
<i>ldq</i>	<p>INTEGER. The leading dimension of the array <i>q</i>; $ldq \geq \max(1, n)$.</p>
<i>indxq</i>	<p>INTEGER. Array, dimension (<i>n</i>). Contains the permutation that separately sorts the two sub-problems in <i>d</i> into ascending order.</p>
<i>rho</i>	<p>REAL for slaed7 /claed7 DOUBLE PRECISION for dlaed7/zlaed7. The subdiagonal element used to create the rank-1 modification.</p>
<i>qstore</i>	<p>REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension (n^2+1). Serves also as output parameter. Stores eigenvectors of submatrices encountered during divide and conquer, packed together. <i>qptr</i> points to beginning of the submatrices.</p>
<i>qptr</i>	<p>INTEGER. Array, dimension $(n+2)$. Serves also as output parameter. List of indices pointing to beginning of submatrices stored in <i>qstore</i>. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.</p>
<i>prmptr, perm, givptr</i>	<p>INTEGER. Arrays, dimension $(n \log_2 n)$ each. The array <i>prmptr</i>(*) contains a list of pointers which indicate where in <i>perm</i> a level's permutation is stored. <i>prmptr</i>(<i>i</i>+1) - <i>prmptr</i>(<i>i</i>) indicates the size of the permutation and also the size of the full, non-deflated problem. The array <i>perm</i>(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock. This parameter can be modified by ? laed8, where it is output.</p>

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(*i*+1) - *givptr*(*i*) indicates the number of Givens rotations.

<i>givcol</i>	INTEGER. Array, dimension (2, $n \log_2 n$). Each pair of numbers indicates a pair of columns to take place in a Givens rotation.
<i>givnum</i>	REAL for slaed7/claed7 DOUBLE PRECISION for dlaed7/zlaed7. Array, dimension (2, $n \log_2 n$). Each number indicates the <i>S</i> value to be used in the corresponding Givens rotation.
<i>iwork</i>	INTEGER. Workspace array, dimension (4 <i>n</i>).
<i>rwork</i>	REAL for claed7 DOUBLE PRECISION for zlaed7. Workspace array, dimension (3 <i>n</i> +2 <i>qsiz</i> * <i>n</i>). Used in complex flavors only.

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.
<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). Contains the permutation that reintegrates the subproblems back into a sorted order, that is, $d(\text{indxq}(i = 1, n))$ will be in the ascending order.
<i>rho</i>	This parameter can be modified by ?1aed8, where it is input/output.
<i>prmptr, perm, givptr</i>	INTEGER. Arrays, dimension ($n \log_2 n$) each. The array <i>prmptr</i> contains an updated list of pointers. The array <i>perm</i> contains an updated permutation. The array <i>givptr</i> contains an updated list of pointers.
<i>givcol</i>	This parameter can be modified by ?1aed8, where it is output.
<i>givnum</i>	This parameter can be modified by ?1aed8, where it is output.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed8

Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.

Syntax

```
call slaed8(  icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2,
w, perm, givptr, givcol, givnum, indxp, indx, info )
```

```
call dlaed8(  icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2,
w, perm, givptr, givcol, givnum, indxp, indx, info )
```

```
call claed8(  k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp, indx,
indxq, perm, givptr, givcol, givnum, info )
```

```
call zlaed8(  k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indxp, indx,
indxq, perm, givptr, givcol, givnum, info )
```

Include Files

- mkl.fi

Description

The routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the *z* vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

<i>icompq</i>	INTEGER. Used with real flavors only. If <i>icompq</i> = 0, compute eigenvalues only. If <i>icompq</i> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array <i>q</i> must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n$.
<i>qsiz</i>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <i>icompq</i> = 1).
<i>d, z</i>	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Arrays, dimension at least $\max(1, n)$ each. The array <i>d</i> (*) contains the eigenvalues of the two submatrices to be combined.

On entry, $z(*)$ contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). The contents of z are destroyed by the updating process.

q REAL for slaed8
DOUBLE PRECISION for dlaed8
COMPLEX for claed8
DOUBLE COMPLEX for zlaed8.

Array

$q(ldq, *)$. The second dimension of q must be at least $\max(1, n)$. On entry, q contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems.

For real flavors, If $icompq = 0$, q is not referenced.

ldq INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.

$ldq2$ INTEGER. The leading dimension of the output array $q2$; $ldq2 \geq \max(1, n)$.

$indxq$ INTEGER. Array, dimension (n).

The permutation that separately sorts the two sub-problems in d into ascending order. Note that elements in the second half of this permutation must first have $cutpnt$ added to their values in order to be accurate.

ρ REAL for slaed8/claed8
DOUBLE PRECISION for dlaed8/zlaed8.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.

d On exit, contains the trailing ($n-k$) updated eigenvalues (those which were deflated) sorted into increasing order.

z On exit, the updating process destroys the contents of z .

q On exit, q contains the trailing ($n-k$) updated eigenvectors (those which were deflated) in its last ($n-k$) columns.

$indxq$ INTEGER. Array, dimension (n).

The permutation of merged eigenvalues set.

ρ On exit, ρ has been modified to the value required by ?laed3.

$d\lambda, w$ REAL for slaed8/claed8
DOUBLE PRECISION for dlaed8/zlaed8.

Arrays, dimension (n) each. The array *dlamda*(*) contains a copy of the first k eigenvalues which will be used by ?laed3 to form the secular equation.

The array *w*(*) will hold the first k values of the final deflation-altered z -vector and will be passed to ?laed3.

q2

REAL for slaed8
 DOUBLE PRECISION for dlaed8
 COMPLEX for claed8
 DOUBLE COMPLEX for zlaed8.

Array

q2(*ldq2*, *). The second dimension of *q2* must be at least $\max(1, n)$.

Contains a copy of the first k eigenvectors which will be used by slaed7/dlaed7 in a matrix multiply (*sgemm/dgemm*) to update the new eigenvectors. For real flavors, If *icomprq* = 0, *q2* is not referenced.

indxp, *indx*

INTEGER. Workspace arrays, dimension (n) each.

The array *indxp*(*) will contain the permutation used to place deflated values of d at the end of the array. On output, *indxp*(1: k) points to the nondeflated d -values and *indxp*($k+1:n$) points to the deflated eigenvalues.

The array *indx*(*) will contain the permutation used to sort the contents of d into ascending order.

perm

INTEGER. Array, dimension (n).

Contains the permutations (from deflation and sorting) to be applied to each eigenblock.

givptr

INTEGER. Contains the number of Givens rotations which took place in this subproblem.

givcol

INTEGER. Array, dimension ($2, n$).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum

REAL for slaed8/claed8
 DOUBLE PRECISION for dlaed8/zlaed8.

Array, dimension ($2, n$).

Each number indicates the S value to be used in the corresponding Givens rotation.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

?laed9

Used by `sstedc/dstedc`. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

```
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

Include Files

- `mkl.fi`

Description

The routine finds the roots of the secular equation, as defined by the values in d , z , and ρ , between $kstart$ and $kstop$. It makes the appropriate calls to `slaed4/dlaed4` and then stores the new matrix of eigenvectors for use in calculating the next level of z vectors.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by <code>slaed4/dlaed4</code> ($k \geq 0$).
$kstart, kstop$	INTEGER. The updated eigenvalues $\lambda(i)$, $kstart \leq i \leq kstop$ are to be computed. $1 \leq kstart \leq kstop \leq k$.
n	INTEGER. The number of rows and columns in the Q matrix. $n \geq k$ (deflation may result in $n > k$).
q	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> . Workspace array, dimension $(ldq, *)$. The second dimension of q must be at least $\max(1, n)$.
ldq	INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.
ρ	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> The value of the parameter in the rank one update equation. $\rho \geq 0$ required.
$dlamda, w$	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> Arrays, dimension (k) each. The first k elements of the array $dlamda(*)$ contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first k elements of the array $w(*)$ contain the components of the deflation-adjusted updating vector.

lds INTEGER. The leading dimension of the output array s ; $lds \geq \max(1, k)$.

Output Parameters

d REAL for slaed9
DOUBLE PRECISION for dlaed9
Array, dimension (n). Elements in $d(i)$ are not referenced for $1 \leq i < kstart$ or $kstop < i \leq n$.

s REAL for slaed9
DOUBLE PRECISION for dlaed9.
Array, dimension ($lds, *$).
The second dimension of s must be at least $\max(1, k)$. Will contain the eigenvectors of the repaired matrix which will be stored for subsequent z vector calculation and multiplied by the previously accumulated eigenvectors to update the system.

dlamda On exit, the value is modified to make sure all $d\lambda(i) - d\lambda(j)$ can be computed with high relative accuracy, barring overflow and underflow.

w Destroyed on exit.

info INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, the eigenvalue did not converge.

?laeda

Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.

Syntax

```
call slaeda( n, tlvs, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum, q, qptr,
z, ztemp, info )
```

```
call dlaeda( n, tlvs, curlvl, curpbm, prmptr, perm, givptr, givcol, givnum, q, qptr,
z, ztemp, info )
```

Include Files

- mkl.fi

Description

The routine ?laeda computes the Z vector corresponding to the merge step in the $curlvl$ -th step of the merge process with $tlvs$ steps for the $curpbm$ -th problem.

Input Parameters

<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>tlvls</i>	INTEGER. The total number of merging levels in the overall divide and conquer tree.
<i>curlvl</i>	INTEGER. The current level in the overall merge routine, $0 \leq \text{curlvl} \leq \text{tlvls}$.
<i>curpbm</i>	INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
<i>prmptr, perm, givptr</i>	<p>INTEGER. Arrays, dimension $(n \log_2 n)$ each.</p> <p>The array <i>prmptr</i>(*) contains a list of pointers which indicate where in <i>perm</i> a level's permutation is stored. <i>prmptr</i>(<i>i</i>+1) - <i>prmptr</i>(<i>i</i>) indicates the size of the permutation and also the size of the full, non-deflated problem.</p> <p>The array <i>perm</i>(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.</p> <p>The array <i>givptr</i>(*) contains a list of pointers which indicate where in <i>givcol</i> a level's Givens rotations are stored. <i>givptr</i>(<i>i</i>+1) - <i>givptr</i>(<i>i</i>) indicates the number of Givens rotations.</p>
<i>givcol</i>	<p>INTEGER. Array, dimension $(2, n \log_2 n)$.</p> <p>Each pair of numbers indicates a pair of columns to take place in a Givens rotation.</p>
<i>givnum</i>	<p>REAL for slaeda</p> <p>DOUBLE PRECISION for dlaeda.</p> <p>Array, dimension $(2, n \log_2 n)$.</p> <p>Each number indicates the <i>S</i> value to be used in the corresponding Givens rotation.</p>
<i>q</i>	<p>REAL for slaeda</p> <p>DOUBLE PRECISION for dlaeda.</p> <p>Array, dimension (n^2).</p> <p>Contains the square eigenblocks from previous levels, the starting positions for blocks are given by <i>qptr</i>.</p>
<i>qptr</i>	<p>INTEGER. Array, dimension $(n+2)$. Contains a list of pointers which indicate where in <i>q</i> an eigenblock is stored. $\text{sqrt}(\text{qptr}(\text{i}+1) - \text{qptr}(\text{i}))$ indicates the size of the block.</p>
<i>ztemp</i>	<p>REAL for slaeda</p> <p>DOUBLE PRECISION for dlaeda.</p> <p>Workspace array, dimension (n).</p>

Output Parameters

<i>z</i>	REAL for slaeda
----------	-----------------

DOUBLE PRECISION for dlaeda.

Array, dimension (n). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

call slaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *wr*, *wi*, *vr*, *vi*, *b*, *ldb*, *work*, *eps3*, *smlnum*, *bignum*, *info*)

call dlaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *wr*, *wi*, *vr*, *vi*, *b*, *ldb*, *work*, *eps3*, *smlnum*, *bignum*, *info*)

call claein(*rightv*, *noinit*, *n*, *h*, *ldh*, *w*, *v*, *b*, *ldb*, *rwork*, *eps3*, *smlnum*, *info*)

call zlaein(*rightv*, *noinit*, *n*, *h*, *ldh*, *w*, *v*, *b*, *ldb*, *rwork*, *eps3*, *smlnum*, *info*)

Include Files

- mkl.fi

Description

The routine ?laein uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (wr,wi) of a real upper Hessenberg matrix H (for real flavors slaein/dlaein) or to the eigenvalue w of a complex upper Hessenberg matrix H (for complex flavors claein/zlaein).

Input Parameters

rightv

LOGICAL.

If *rightv* = .TRUE., compute right eigenvector;

if *rightv* = .FALSE., compute left eigenvector.

noinit

LOGICAL.

If *noinit* = .TRUE., no initial vector is supplied in (*vr,vi*) or in *v* (for complex flavors);

if *noinit* = .FALSE., initial vector is supplied in (*vr,vi*) or in *v* (for complex flavors).

n

INTEGER. The order of the matrix H ($n \geq 0$).

h

REAL for slaein

DOUBLE PRECISION for dlaein

COMPLEX for claein

DOUBLE COMPLEX for `zlaein`.

Array $h(ldh, *)$.

The second dimension of h must be at least $\max(1, n)$. Contains the upper Hessenberg matrix H .

ldh INTEGER. The leading dimension of the array h ; $ldh \geq \max(1, n)$.

wr, wi REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

The real and imaginary parts of the eigenvalue of H whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).

w COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

The eigenvalue of H whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).

vr, vi REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

Arrays, dimension (n) each. Used for real flavors only. On entry, if `noinit = .FALSE.` and $wi = 0.0$, vr must contain a real starting vector for inverse iteration using the real eigenvalue wr ;

if `noinit = .FALSE.` and $wi \neq 0.0$, vr and vi must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (wr, wi) ; otherwise vr and vi need not be set.

v COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

Array, dimension (n). Used for complex flavors only. On entry, if `noinit = .FALSE.`, v must contain a starting vector for inverse iteration; otherwise v need not be set.

b REAL for `slaein`

DOUBLE PRECISION for `dlaein`

COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

Workspace array $b(l db, *)$. The second dimension of b must be at least $\max(1, n)$.

ldb INTEGER. The leading dimension of the array b ;

$ldb \geq n+1$ for real flavors;

$ldb \geq \max(1, n)$ for complex flavors.

work REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

Workspace array, dimension (n).

Used for real flavors only.

rwork

REAL for `claein`

DOUBLE PRECISION for `zlaein`.

Workspace array, dimension (*n*).

Used for complex flavors only.

eps3, smlnum

REAL for `slaein/claein`

DOUBLE PRECISION for `dlaein/zlaein`.

eps3 is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots.

smlnum is a machine-dependent value close to underflow threshold. A suggested value for *smlnum* is `slamch('s') * (n/slamch('p'))` for `slaein/claein` or `dlamch('s') * (n/dlamch('p'))` for `dlaein/zlaein`. See [lamch](#).

bignum

REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

bignum is a machine-dependent value close to overflow threshold. Used for real flavors only. A suggested value for *bignum* is `1 / slamch('s')` for `slaein/claein` or `1 / dlamch('s')` for `dlaein/zlaein`.

Output Parameters

vr, vi

On exit, if $w_i = 0.0$ (real eigenvalue), *vr* contains the computed real eigenvector; if $w_i \neq 0.0$ (complex eigenvalue), *vr* and *vi* contain the real and imaginary parts of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x,y*) is taken to be $|x| + |y|$.

vi is not referenced if $w_i = 0.0$.

v

On exit, *v* contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (*x,y*) is taken to be $|x| + |y|$.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, inverse iteration did not converge. For real flavors, *vr* is set to the last iterate, and so is *vi*, if $w_i \neq 0.0$. For complex flavors, *v* is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

```
call slaev2( a, b, c, rt1, rt2, cs1, sn1 )
```

```
call dlaev2( a, b, c, rt1, rt2, cs1, sn1 )
call claev2( a, b, c, rt1, rt2, cs1, sn1 )
call zlaev2( a, b, c, rt1, rt2, cs1, sn1 )
```

Include Files

- mkl.fi

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \text{ (for } slaev2/dlaev2) \text{ or Hermitian matrix } \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix}$$

(for claev2/zlaev2).

On return, *rt1* is the eigenvalue of larger absolute value, *rt2* of smaller absolute value, and (*cs1*, *sn1*) is the unit right eigenvector for *rt1*, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for slaev2/dlaev2),

or

$$\begin{bmatrix} cs1 & \text{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\text{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for claev2/zlaev2).

Input Parameters

a, *b*, *c* REAL for slaev2
 DOUBLE PRECISION for dlaev2
 COMPLEX for claev2
 DOUBLE COMPLEX for zlaev2.
 Elements of the input matrix.

Output Parameters

rt1, *rt2* REAL for slaev2/claev2
 DOUBLE PRECISION for dlaev2/zlaev2.
 Eigenvalues of larger and smaller absolute value, respectively.

cs1 REAL for slaev2/claev2
DOUBLE PRECISION for dlaev2/zlaev2.

sn1 REAL for slaev2
DOUBLE PRECISION for dlaev2
COMPLEX for claev2
DOUBLE COMPLEX for zlaev2.

The vector (*cs1*, *sn1*) is the unit right eigenvector for *rt1*.

Application Notes

rt1 is accurate to a few ulps barring over/underflow. *rt2* may be inaccurate if there is massive cancellation in the determinant $a*c-b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute *rt2* accurately in all cases. *cs1* and *sn1* are accurate to a few ulps barring over/underflow. Overflow is possible only if *rt1* is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds *underflow_threshold* / *macheps*.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
call dlaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Include Files

- mkl.fi

Description

The routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

wantq LOGICAL.
If *wantq* = .TRUE., accumulate the transformation in the matrix Q ;
If *wantq* = .FALSE., do not accumulate the transformation.

n INTEGER. The order of the matrix T ($n \geq 0$).

t, q REAL for slaexc
DOUBLE PRECISION for dlaexc

Arrays:

$t(ldt,*)$ contains on entry the upper quasi-triangular matrix T , in Schur canonical form.

The second dimension of t must be at least $\max(1, n)$.

$q(ldq,*)$ contains on entry, if $wantq = .TRUE.$, the orthogonal matrix Q . If $wantq = .FALSE.$, q is not referenced. The second dimension of q must be at least $\max(1, n)$.

ldt INTEGER. The leading dimension of t ; at least $\max(1, n)$.

ldq INTEGER. The leading dimension of q ;
If $wantq = .FALSE.$, then $ldq \geq 1$.
If $wantq = .TRUE.$, then $ldq \geq \max(1, n)$.

j1 INTEGER. The index of the first row of the first block T_{11} .

n1 INTEGER. The order of the first block T_{11}
($n1 = 0, 1, \text{ or } 2$).

n2 INTEGER. The order of the second block T_{22}
($n2 = 0, 1, \text{ or } 2$).

work REAL for slaexc;
DOUBLE PRECISION for dlaexc.
Workspace array, DIMENSION (n).

Output Parameters

t On exit, the updated matrix T , again in Schur canonical form.

q On exit, if $wantq = .TRUE.$, the updated matrix Q .

info INTEGER.
If $info = 0$, the execution is successful.
If $info = 1$, the transformed matrix T would be too far from Schur form; the blocks are not swapped and T and Q are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

```
call dlag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a 2 x 2 generalized eigenvalue problem $A - w*B$, with scaling as necessary to avoid over-/underflow. The scaling factor, s , results in a modified eigenvalue equation

$$s*A - w*B,$$

where s is a non-negative scaling factor chosen so that w , $w*B$, and $s*A$ do not overflow and, if possible, do not underflow, either.

Input Parameters

<i>a, b</i>	REAL for <code>slag2</code> DOUBLE PRECISION for <code>dlag2</code> Arrays: <i>a(lda,2)</i> contains, on entry, the 2 x 2 matrix <i>A</i> . It is assumed that its 1-norm is less than $1/safmin$. Entries less than $\sqrt{safmin} * \text{norm}(A)$ are subject to being treated as zero. <i>b(lb,2)</i> contains, on entry, the 2 x 2 upper triangular matrix <i>B</i> . It is assumed that the one-norm of <i>B</i> is less than $1/safmin$. The diagonals should be at least \sqrt{safmin} times the largest element of <i>B</i> (in absolute value); if a diagonal is smaller than that, then $\pm \sqrt{safmin}$ will be used instead of that diagonal.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq 2$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq 2$.
<i>safmin</i>	REAL for <code>slag2</code> ; DOUBLE PRECISION for <code>dlag2</code> . The smallest positive number such that $1/safmin$ does not overflow. (This should always be <code>?lamch('S')</code> - it is an argument in order to avoid having to call <code>?lamch</code> frequently.)

Output Parameters

<i>scale1</i>	REAL for <code>slag2</code> ; DOUBLE PRECISION for <code>dlag2</code> . A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are $(wr1 \pm wii)/scale1$ (which may lie outside the exponent range of the machine), $scale1=scale2$, and <i>scale1</i> will always be positive. If the eigenvalues are real, then the first (real) eigenvalue is $wr1/scale1$, but this may overflow or underflow, and in fact, <i>scale1</i> may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.
<i>scale2</i>	REAL for <code>slag2</code> ; DOUBLE PRECISION for <code>dlag2</code> .

A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then $scale2=scale1$. If the eigenvalues are real, then the second (real) eigenvalue is $wr2/scale2$, but this may overflow or underflow, and in fact, $scale2$ may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1 REAL for `slag2`;
DOUBLE PRECISION for `dlag2`.

If the eigenvalue is real, then *wr1* is $scale1$ times the eigenvalue closest to the (2,2) element of $A*inv(B)$.

If the eigenvalue is complex, then $wr1=wr2$ is $scale1$ times the real part of the eigenvalues.

wr2 REAL for `slag2`;
DOUBLE PRECISION for `dlag2`.

If the eigenvalue is real, then *wr2* is $scale2$ times the other eigenvalue. If the eigenvalue is complex, then $wr1=wr2$ is $scale1$ times the real part of the eigenvalues.

wi REAL for `slag2`;
DOUBLE PRECISION for `dlag2`.

If the eigenvalue is real, then *wi* is zero. If the eigenvalue is complex, then *wi* is $scale1$ times the imaginary part of the eigenvalues. *wi* will always be non-negative.

slags2

Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call dlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call clags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call zlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
```

Include Files

- `mkl.fi`

Description

For real flavors, the routine computes 2-by-2 orthogonal matrices U , V and Q , such that if `upper = .TRUE.`, then

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} & a_{19} & a_{20} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} & a_{29} & a_{30} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} & a_{39} & a_{40} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} & a_{49} & a_{50} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} & a_{59} & a_{60} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} & a_{69} & a_{70} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} & a_{79} & a_{80} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} & a_{89} & a_{90} \\ a_{91} & a_{92} & a_{93} & a_{94} & a_{95} & a_{96} & a_{97} & a_{98} & a_{99} & a_{100} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} & b_{17} & b_{18} & b_{19} & b_{20} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} & b_{27} & b_{28} & b_{29} & b_{30} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} & b_{37} & b_{38} & b_{39} & b_{40} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} & b_{47} & b_{48} & b_{49} & b_{50} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} & b_{56} & b_{57} & b_{58} & b_{59} & b_{60} \\ b_{61} & b_{62} & b_{63} & b_{64} & b_{65} & b_{66} & b_{67} & b_{68} & b_{69} & b_{70} \\ b_{71} & b_{72} & b_{73} & b_{74} & b_{75} & b_{76} & b_{77} & b_{78} & b_{79} & b_{80} \\ b_{81} & b_{82} & b_{83} & b_{84} & b_{85} & b_{86} & b_{87} & b_{88} & b_{89} & b_{90} \\ b_{91} & b_{92} & b_{93} & b_{94} & b_{95} & b_{96} & b_{97} & b_{98} & b_{99} & b_{100} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} & c_{17} & c_{18} & c_{19} & c_{20} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} & c_{27} & c_{28} & c_{29} & c_{30} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & c_{37} & c_{38} & c_{39} & c_{40} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} & c_{48} & c_{49} & c_{50} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} & c_{58} & c_{59} & c_{60} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} & c_{68} & c_{69} & c_{70} \\ c_{71} & c_{72} & c_{73} & c_{74} & c_{75} & c_{76} & c_{77} & c_{78} & c_{79} & c_{80} \\ c_{81} & c_{82} & c_{83} & c_{84} & c_{85} & c_{86} & c_{87} & c_{88} & c_{89} & c_{90} \\ c_{91} & c_{92} & c_{93} & c_{94} & c_{95} & c_{96} & c_{97} & c_{98} & c_{99} & c_{100} \end{pmatrix}$$

and

$$V^T * B * Q = V^T * \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if `upper = .FALSE.`, then

$$U^T * A * Q = U^T * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$\begin{bmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

The rows of the transformed A and B are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z^T denotes the transpose of Z .

For complex flavors, the routine computes 2-by-2 unitary matrices U , V and Q , such that if `upper = .TRUE.`, then

$$\begin{bmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if `upper = .FALSE.`, then

$$U^H * A * Q = U^H * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed A and B are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu^H & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv^H & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq^H & csq \end{bmatrix}$$

Input Parameters

<i>upper</i>	LOGICAL. If <i>upper</i> = .TRUE., the input matrices A and B are upper triangular; If <i>upper</i> = .FALSE., the input matrices A and B are lower triangular.
<i>a1, a3</i>	REAL for <i>slags2</i> and <i>clags2</i> DOUBLE PRECISION for <i>dlags2</i> and <i>zlags2</i>
<i>a2</i>	REAL for <i>slags2</i> DOUBLE PRECISION for <i>dlags2</i> COMPLEX for <i>clags2</i> COMPLEX*16 for <i>zlags2</i> On entry, <i>a1</i> , <i>a2</i> and <i>a3</i> are elements of the input 2-by-2 upper (lower) triangular matrix A .
<i>b1, b3</i>	REAL for <i>slags2</i> and <i>clags2</i> DOUBLE PRECISION for <i>dlags2</i> and <i>zlags2</i>
<i>b2</i>	REAL for <i>slags2</i> DOUBLE PRECISION for <i>dlags2</i> COMPLEX for <i>clags2</i> COMPLEX*16 for <i>zlags2</i> On entry, <i>b1</i> , <i>b2</i> and <i>b3</i> are elements of the input 2-by-2 upper (lower) triangular matrix B .

Output Parameters

<i>csu</i>	<p>REAL for <i>slags2</i> and <i>clags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i> and <i>zlags2</i></p> <p>Element of the desired orthogonal matrix <i>U</i>.</p>
<i>snu</i>	<p>REAL for <i>slags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i></p> <p>Element of the desired orthogonal matrix <i>U</i>.</p> <p>COMPLEX for <i>clags2</i></p> <p>COMPLEX*16 for <i>zlags2</i></p>
<i>csv</i>	<p>REAL for <i>slags2</i> and <i>clags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i> and <i>zlags2</i></p> <p>Element of the desired orthogonal matrix <i>V</i>.</p>
<i>snv</i>	<p>REAL for <i>slags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i></p> <p>COMPLEX for <i>clags2</i></p> <p>COMPLEX*16 for <i>zlags2</i></p> <p>Element of the desired orthogonal matrix <i>V</i>.</p>
<i>csq</i>	<p>REAL for <i>slags2</i> and <i>clags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i> and <i>zlags2</i></p> <p>Element of the desired orthogonal matrix <i>Q</i>.</p>
<i>snq</i>	<p>REAL for <i>slags2</i></p> <p>DOUBLE PRECISION for <i>dlags2</i></p> <p>Element of the desired orthogonal matrix <i>Q</i>.</p> <p>COMPLEX for <i>clags2</i></p> <p>COMPLEX*16 for <i>zlags2</i></p>

?lagtf

Computes an LU factorization of a matrix $T-\lambda I$, where T is a general tridiagonal matrix, and λ is a scalar, using partial pivoting with row interchanges.

Syntax

```
call slagtf( n, a, lambda, b, c, tol, d, in, info )
call dlagtf( n, a, lambda, b, c, tol, d, in, info )
```

Include Files

- `mkl.fi`

Description

The routine factorizes the matrix $(T - \lambda I)$, where T is an n -by- n tridiagonal matrix and λ is a scalar, as

$$T - \lambda I = P^* L^* U,$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column. The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling. The parameter λ is included in the routine so that `?slagtf` may be used, in conjunction with `?lagts`, to obtain eigenvectors of T by inverse iteration.

Input Parameters

n	INTEGER. The order of the matrix T ($n \geq 0$).
a, b, c	REAL for <code>slagtf</code> DOUBLE PRECISION for <code>dlagtf</code> Arrays, dimension $a(n), b(n-1), c(n-1)$: On entry, $a(*)$ must contain the diagonal elements of the matrix T . On entry, $b(*)$ must contain the $(n-1)$ super-diagonal elements of T . On entry, $c(*)$ must contain the $(n-1)$ sub-diagonal elements of T .
tol	REAL for <code>slagtf</code> DOUBLE PRECISION for <code>dlagtf</code> On entry, a relative tolerance used to indicate whether or not the matrix $(T - \lambda I)$ is nearly singular. tol should normally be chosen as approximately the largest relative error in the elements of T . For example, if the elements of T are correct to about 4 significant figures, then tol should be set to about 5×10^{-4} . If tol is supplied as less than ϵ , where ϵ is the relative machine precision, then the value ϵ is used in place of tol .

Output Parameters

a	On exit, a is overwritten by the n diagonal elements of the upper triangular matrix U of the factorization of T .
b	On exit, b is overwritten by the $n-1$ super-diagonal elements of the matrix U of the factorization of T .
c	On exit, c is overwritten by the $n-1$ sub-diagonal elements of the matrix L of the factorization of T .
d	REAL for <code>slagtf</code> DOUBLE PRECISION for <code>dlagtf</code> Array, dimension $(n-2)$. On exit, d is overwritten by the $n-2$ second super-diagonal elements of the matrix U of the factorization of T .
in	INTEGER.

Array, dimension (n).

On exit, in contains details of the permutation matrix p . If an interchange occurred at the k -th step of the elimination, then $in(k) = 1$, otherwise $in(k) = 0$. The element $in(n)$ returns the smallest positive integer j such that

$$\text{abs}(u(j,j)) \leq \text{norm}((T - \text{lambda}*I)(j)) * \text{tol},$$

where $\text{norm}(A(j))$ denotes the sum of the absolute values of the j -th row of the matrix A .

If no such j exists then $in(n)$ is returned as zero. If $in(n)$ is returned as positive, then a diagonal element of U is small, indicating that $(T - \text{lambda}*I)$ is singular or nearly singular.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -k$, the k -th parameter had an illegal value.

?lagtm

Performs a matrix-matrix product of the form $C = \alpha*A*B + \beta*C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
```

Include Files

- mkl.fi

Description

The routine performs a matrix-vector product of the form:

$$B := \alpha*A*X + \beta*B$$

where A is a tridiagonal matrix of order n , B and X are n -by- $nrhs$ matrices, and α and β are real scalars, each of which may be 0., 1., or -1.

Input Parameters

$trans$

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N'$, then $B := \alpha*A*X + \beta*B$ (no transpose);

If $trans = 'T'$, then $B := \alpha*A^T*X + \beta*B$ (transpose);

If $trans = 'C'$, then $B := \alpha*A^H*X + \beta*B$ (conjugate transpose)

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides, i.e., the number of columns in <i>X</i> and <i>B</i> ($nrhs \geq 0$).
<i>alpha, beta</i>	REAL for slagtm/clagtm DOUBLE PRECISION for dlagtm/zlagtm Specify the scalars <i>alpha</i> and <i>beta</i> respectively. <i>alpha</i> must be 0., 1., or -1.; otherwise, it is assumed to be 0. <i>beta</i> must be 0., 1., or -1.; otherwise, it is assumed to be 1.
<i>dl, d, du</i>	REAL for slagtm DOUBLE PRECISION for dlagtm COMPLEX for clagtm DOUBLE COMPLEX for zlagtm. Arrays: <i>dl</i> ($n - 1$), <i>d</i> (n), <i>du</i> ($n - 1$). The array <i>dl</i> contains the ($n - 1$) sub-diagonal elements of <i>T</i> . The array <i>d</i> contains the n diagonal elements of <i>T</i> . The array <i>du</i> contains the ($n - 1$) super-diagonal elements of <i>T</i> .
<i>x, b</i>	REAL for slagtm DOUBLE PRECISION for dlagtm COMPLEX for clagtm DOUBLE COMPLEX for zlagtm. Arrays: <i>x</i> (<i>ldx</i> ,*) contains the n -by- <i>nrhs</i> matrix <i>X</i> . The second dimension of <i>x</i> must be at least $\max(1, nrhs)$. <i>b</i> (<i>ldb</i> ,*) contains the n -by- <i>nrhs</i> matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldx</i>	INTEGER. The leading dimension of the array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the matrix expression $B := \alpha A * X + \beta B$

?lagts

Solves the system of equations $(T - \lambda I) * x = y$ or $(T - \lambda I)^T * x = y$, where *T* is a general tridiagonal matrix and λ is a scalar, using the LU factorization computed by ?lagtf.

Syntax

call slagts(*job*, *n*, *a*, *b*, *c*, *d*, *in*, *y*, *tol*, *info*)

call `dlagts(job, n, a, b, c, d, in, y, tol, info)`

Include Files

- `mkl.fi`

Description

The routine may be used to solve for x one of the systems of equations:

$$(T - \lambda I)x = y \text{ or } (T - \lambda I)^T x = y,$$

where T is an n -by- n tridiagonal matrix, following the factorization of $(T - \lambda I)$ as

$$T - \lambda I = P^*L^*U,$$

computed by the routine `?lagtf`.

The choice of equation to be solved is controlled by the argument `job`, and in each case there is an option to perturb zero or very small diagonal elements of U , this option being intended for use in applications such as inverse iteration.

Input Parameters

<code>job</code>	<p>INTEGER. Specifies the job to be performed by <code>?lagts</code> as follows:</p> <ul style="list-style-type: none"> = 1: The equations $(T - \lambda I)x = y$ are to be solved, but diagonal elements of U are not to be perturbed. = -1: The equations $(T - \lambda I)x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <code>tol</code> below. = 2: The equations $(T - \lambda I)^T x = y$ are to be solved, but diagonal elements of U are not to be perturbed. = -2: The equations $(T - \lambda I)^T x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <code>tol</code> below.
<code>n</code>	<p>INTEGER. The order of the matrix T ($n \geq 0$).</p>
<code>a, b, c, d</code>	<p>REAL for <code>slagts</code></p> <p>DOUBLE PRECISION for <code>dlagts</code></p> <p>Arrays, dimension $a(n)$, $b(n-1)$, $c(n-1)$, $d(n-2)$:</p> <p>On entry, <code>a(*)</code> must contain the diagonal elements of U as returned from <code>?lagtf</code>.</p> <p>On entry, <code>b(*)</code> must contain the first super-diagonal elements of U as returned from <code>?lagtf</code>.</p> <p>On entry, <code>c(*)</code> must contain the sub-diagonal elements of L as returned from <code>?lagtf</code>.</p> <p>On entry, <code>d(*)</code> must contain the second super-diagonal elements of U as returned from <code>?lagtf</code>.</p>
<code>in</code>	<p>INTEGER.</p> <p>Array, dimension (n).</p>

On entry, *in* (*) must contain details of the matrix *p* as returned from ?lagtf.

y REAL for slagts
DOUBLE PRECISION for dlagts
Array, dimension (*n*). On entry, the right hand side vector *y*.

tol REAL for slagtf
DOUBLE PRECISION for dlagtf.
On entry, with *job* < 0, *tol* should be the minimum perturbation to be made to very small diagonal elements of *U*. *tol* should normally be chosen as about *eps**norm(*U*), where *eps* is the relative machine precision, but if *tol* is supplied as non-positive, then it is reset to *eps**max(abs(u(*i*,*j*))). If *job* > 0 then *tol* is not referenced.

Output Parameters

y On exit, *y* is overwritten by the solution vector *x*.

tol On exit, *tol* is changed as described in *Input Parameters* section above, only if *tol* is non-positive on entry. Otherwise *tol* is unchanged.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* > 0, overflow would occur when computing the *i*th element of the solution vector *x*. This can only occur when *job* is supplied as positive and either means that a diagonal element of *U* is very small, or that the elements of the right-hand side vector *y* are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A*,*B*) where *B* is upper triangular.

Syntax

```
call slagv2( a, lda, b, ldb, alphas, alphai, beta, csl, snl, csr, snr )
call dlagv2( a, lda, b, ldb, alphas, alphai, beta, csl, snl, csr, snr )
```

Include Files

- mkl.fi

Description

The routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A*,*B*) where *B* is upper triangular. The routine computes orthogonal (rotation) matrices given by *csl*, *snl* and *csr*, *snr* such that:

- 1) if the pencil (*A*,*B*) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

2) if the pencil (A,B) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, b REAL for slagv2
DOUBLE PRECISION for dlagv2
Arrays:
a(lda,2) contains the 2-by-2 matrix *A*;
b(ldb,2) contains the upper triangular 2-by-2 matrix *B*.

lda INTEGER. The leading dimension of the array *a*;
lda ≥ 2.

ldb INTEGER. The leading dimension of the array *b*;
ldb ≥ 2.

Output Parameters

a On exit, *a* is overwritten by the "A-part" of the generalized Schur form.

b On exit, *b* is overwritten by the "B-part" of the generalized Schur form.

alphar, alphai, beta REAL for slagv2
DOUBLE PRECISION for dlagv2.
Arrays, dimension (2) each.
 $(\text{alphar}(k) + i \cdot \text{alphai}(k)) / \text{beta}(k)$ are the eigenvalues of the pencil (A,B) , $k=1,2$ and $i = \text{sqrt}(-1)$.

Note that $\beta(k)$ may be zero.

csl, snl

REAL for `slagv2`

DOUBLE PRECISION for `dlagv2`

The cosine and sine of the left rotation matrix, respectively.

csr, snr

REAL for `slagv2`

DOUBLE PRECISION for `dlagv2`

The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.

Syntax

```
call slahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
```

```
call dlahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
```

```
call clahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

```
call zlahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

Include Files

- `mkl.fi`

Description

The routine is an auxiliary routine called by `?hseqr` to update the eigenvalues and Schur decomposition already computed by `?hseqr`, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*.

Input Parameters

wantt

LOGICAL.

If *wantt* = `.TRUE.`, the full Schur form *T* is required;

If *wantt* = `.FALSE.`, eigenvalues only are required.

wantz

LOGICAL.

If *wantz* = `.TRUE.`, the matrix of Schur vectors *Z* is required;

If *wantz* = `.FALSE.`, Schur vectors are not required.

n

INTEGER. The order of the matrix *H* ($n \geq 0$).

ilo, ihi

INTEGER.

It is assumed that *h* is already upper quasi-triangular in rows and columns *ihi*+1:*n*, and that $h(i\text{lo}, i\text{lo}-1) = 0$ (unless $i\text{lo} = 1$). The routine `?lahqr` works primarily with the Hessenberg submatrix in rows and columns *ilo* to *ihi*, but applies transformations to all of *h* if *wantt* = `.TRUE.`

Constraints:

$1 \leq ilo \leq \max(1, ihi); ihi \leq n.$

h, z

REAL for `slahqr`

DOUBLE PRECISION for `dlahqr`

COMPLEX for `clahqr`

DOUBLE COMPLEX for `zlahqr`.

Arrays:

h(ldh,)* contains the upper Hessenberg matrix *h*.

The second dimension of *h* must be at least $\max(1, n)$.

z(ldz,)*

If *wantz* = `.TRUE.`, then, on entry, *z* must contain the current matrix *z* of transformations accumulated by `?hseqr`. The second dimension of *z* must be at least $\max(1, n)$

If *wantz* = `.FALSE.`, then *z* is not referenced..

ldh

INTEGER. The leading dimension of *h*; at least $\max(1, n)$.

ldz

INTEGER. The leading dimension of *z*; at least $\max(1, n)$.

iloz, ihiz

INTEGER. Specify the rows of *z* to which transformations must be applied if *wantz* = `.TRUE.`.

$1 \leq iloz \leq ilo; ihi \leq ihiz \leq n.$

Output Parameters

h

On exit, if *info*= 0 and *wantt* = `.TRUE.`, then,

- for `slahqr/dlahqr`, *h* is upper quasi-triangular in rows and columns *ilo:ihi* with any 2-by-2 diagonal blocks in standard form.
- for `clahqr/zlahqr`, *h* is upper triangular in rows and columns *ilo:ihi*.

If *info*= 0 and *wantt* = `.FALSE.`, the contents of *h* are unspecified on exit. If *info* is positive, see description of *info* for the output state of *h*.

wr, wi

REAL for `slahqr`

DOUBLE PRECISION for `dlahqr`

Arrays, DIMENSION at least $\max(1, n)$ each. Used with real flavors only. The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with $wi(i) > 0$ and $wi(i+1) < 0$.

If *wantt* = `.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with $wr(i) = h(i, i)$, and, if *h*(*i*:*i*+1, *i*:*i*+1) is a 2-by-2 diagonal block, $wi(i) = \text{sqrt}(h(i+1, i)*h(i, i+1))$ and $wi(i+1) = -wi(i)$.

w COMPLEX for `clahqr`
DOUBLE COMPLEX for `zlahqr`.

Array, DIMENSION at least $\max(1, n)$. Used with complex flavors only. The computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *w*.

If *wantt* = `.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with $w(i) = h(i, i)$.

z If *wantz* = `.TRUE.`, then, on exit *z* has been updated; transformations are applied only to the submatrix $z(iloz:i hiz, ilo:i hi)$.

info INTEGER.

If *info* = 0, the execution is successful.

With *info* > 0,

- if *info* = *i*, `?lahqr` failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30 iterations per eigenvalue; elements $i+1:i hi$ of *wr* and *wi* (for `slahqr/dlahqr`) or *w* (for `clahqr/zlahqr`) contain those eigenvalues which have been successfully computed.
- if *wantt* is `.FALSE.`, then on exit the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.
- if *wantt* is `.TRUE.`, then on exit
(initial value of *h*)**u* = *u**(final value of *h*), (*)
where *u* is an orthogonal matrix. The final value of *h* is upper Hessenberg and triangular in rows and columns *info*+1 through *ihi*.
- if *wantz* is `.TRUE.`, then on exit
(final value of *z*) = (initial value of *z*)* *u*,
where *u* is an orthogonal matrix in (*) regardless of the value of *wantt*.

?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.

Syntax

```
call slahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- `mkl.fi`

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q^T A Q$ for real flavors, or $Q^H A Q$ for complex flavors. The routine returns the matrices V and T which determine Q as a block reflector $I - V^* T^* V^T$ (for real flavors) or $I - V^* T^* V^H$ (for complex flavors), and also the matrix $Y = A^* V^* T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau v v^H \text{ for complex flavors, or}$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an obsolete auxiliary routine. Please use the new routine `?lahr2` instead.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
nb	INTEGER. The number of columns to be reduced.
a	REAL for <code>slahrd</code> DOUBLE PRECISION for <code>dlahrd</code> COMPLEX for <code>clahrd</code> DOUBLE COMPLEX for <code>zlahrd</code> . Array $a(lda, n-k+1)$ contains the n -by- $(n-k+1)$ general matrix A to be reduced.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldt	INTEGER. The leading dimension of the output array t ; must be at least $\max(1, nb)$.
ldy	INTEGER. The leading dimension of the output array y ; must be at least $\max(1, n)$.

Output Parameters

a	On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced matrix; the elements below the k -th subdiagonal, with the array τ , represent the matrix Q as a product of elementary reflectors. The other columns of a are unchanged. See <i>Application Notes</i> below.
τ	REAL for <code>slahrd</code> DOUBLE PRECISION for <code>dlahrd</code> COMPLEX for <code>clahrd</code> DOUBLE COMPLEX for <code>zlahrd</code> .

Array, DIMENSION (nb).

Contains scalar factors of the elementary reflectors.

t, y

REAL for `slahrd`

DOUBLE PRECISION for `dlahrd`

COMPLEX for `clahrd`

DOUBLE COMPLEX for `zlahrd`.

Arrays, dimension $t(ldt, nb)$, $y(ldy, nb)$.

The array t contains upper triangular matrix T .

The array y contains the n -by- nb matrix Y .

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $a(i+k+1:n, i)$ and τ is stored in $\tau(i)$.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V^* T^* V^T) * (A - Y^* V^T)$ for real flavors, or

$A := (I - V^* T^* V^H) * (A - Y^* V^H)$ for complex flavors.

The contents of A on exit are illustrated by the following example with $n = 7$, $k = 3$ and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

See Also

[?lahr2](#)

[?lahr2](#)

Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .

Syntax

```
call slahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- mkl.fi

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q^T A Q$ for real flavors, or $Q^H A Q$ for complex flavors. The routine returns the matrices V and T which determine Q as a block reflector $I - V^* T^* V^T$ (for real flavors) or $I - V^* T^* V^H$ (for real flavors), and also the matrix $Y = A^* V^* T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau * v * v^H \text{ for complex flavors}$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an auxiliary routine called by ?gehrd.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero ($k < n$).
nb	INTEGER. The number of columns to be reduced.
a	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Array, DIMENSION ($lda, n-k+1$) contains the n -by- $(n-k+1)$ general matrix A to be reduced.
lda	INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.
ldt	INTEGER. The leading dimension of the output array t ; $ldt \geq nb$.
ldy	INTEGER. The leading dimension of the output array y ; $ldy \geq n$.

Output Parameters

<i>a</i>	On exit, the elements on and above the <i>k</i> -th subdiagonal in the first <i>nb</i> columns are overwritten with the corresponding elements of the reduced matrix; the elements below the <i>k</i> -th subdiagonal, with the array <i>tau</i> , represent the matrix <i>Q</i> as a product of elementary reflectors. The other columns of <i>a</i> are unchanged. See <i>Application Notes</i> below.
<i>tau</i>	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Array, DIMENSION (<i>nb</i>). Contains scalar factors of the elementary reflectors.
<i>t, y</i>	REAL for slahr2 DOUBLE PRECISION for dlahr2 COMPLEX for clahr2 DOUBLE COMPLEX for zlahr2. Arrays, dimension <i>t</i> (<i>ldt, nb</i>), <i>y</i> (<i>ldy, nb</i>). The array <i>t</i> contains upper triangular matrix <i>T</i> . The array <i>y</i> contains the <i>n</i> -by- <i>nb</i> matrix <i>Y</i> .

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $a(i+k+1:n, i)$ and *tau* is stored in $tau(i)$.

The elements of the vectors *v* together form the $(n-k+1)$ -by-*nb* matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V^* T^* V^T) * (A - Y^* V^T)$ for real flavors, or

$A := (I - V^* T^* V^H) * (A - Y^* V^H)$ for complex flavors.

The contents of *A* on exit are illustrated by the following example with $n = 7$, $k = 3$ and $nb = 2$:

$$\begin{bmatrix} a & a & a & a & a \\ a & a & a & a & a \\ a & a & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
```

Include Files

- mkl.fi

Description

The routine ?laic1 applies one step of incremental condition estimation in its simplest version.

Let x , $\|x\|_2 = 1$ (where $\|a\|_2$ denotes the 2-norm of a), be an approximate singular vector of an j -by- j lower triangular matrix L , such that

$$\|Lx\|_2 = sest$$

Then ?laic1 computes $sestpr$, s , c such that the vector

$$xhat = \begin{bmatrix} s*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$Lhat = \begin{bmatrix} L & 0 \\ w^x & gamma \end{bmatrix} \text{ (for complex flavors), or}$$

$$Lhat = \begin{bmatrix} L & 0 \\ w^T & gamma \end{bmatrix} \text{ (for real flavors), in the sense that}$$

$$\|Lhat*xhat\|_2 = sestpr.$$

Depending on job , an estimate for the largest or smallest singular value is computed.

For real flavors, $[sc]^T$ and $sestpr^2$ is an eigenpair of the system

$$\text{diag}(sest*sest, 0) + [alpha \ gamma] * \begin{bmatrix} alpha \\ gamma \end{bmatrix}$$

where $alpha = x^T*w$.

For complex flavors, $[sc]^H$ and $sestpr^2$ is an eigenpair of the system

$$\text{diag}(s_{est} * s_{est}, 0) + [\alpha \ \gamma] * \begin{bmatrix} \text{conjg}(\alpha) \\ \text{conjg}(\gamma) \end{bmatrix}$$

where $\alpha = x^H * w$.

Input Parameters

job INTEGER.
 If *job* =1, an estimate for the largest singular value is computed;
 If *job* =2, an estimate for the smallest singular value is computed;

j INTEGER. Length of *x* and *w*.

x, w REAL for slaic1
 DOUBLE PRECISION for dlaic1
 COMPLEX for claic1
 DOUBLE COMPLEX for zlaic1.
 Arrays, dimension (*j*) each. Contain vectors *x* and *w*, respectively.

sest REAL for slaic1/claic1;
 DOUBLE PRECISION for dlaic1/zlaic1.
 Estimated singular value of *j*-by-*j* matrix *L*.

gamma REAL for slaic1
 DOUBLE PRECISION for dlaic1
 COMPLEX for claic1
 DOUBLE COMPLEX for zlaic1.
 The diagonal element *gamma*.

Output Parameters

sestpr REAL for slaic1/claic1;
 DOUBLE PRECISION for dlaic1/zlaic1.
 Estimated singular value of (*j*+1)-by-(*j*+1) matrix *Lhat*.

s, c REAL for slaic1
 DOUBLE PRECISION for dlaic1
 COMPLEX for claic1
 DOUBLE COMPLEX for zlaic1.
 Sine and cosine needed in forming *xhat*.

?lakf2

Forms a matrix containing Kronecker products between the given matrices.

Syntax

```
call slakf2( m, n, a, lda, b, d, e, z, ldz )
call dlakf2( m, n, a, lda, b, d, e, z, ldz )
call clakf2( m, n, a, lda, b, d, e, z, ldz )
call zlakf2( m, n, a, lda, b, d, e, z, ldz )
```

Include Files

- mkl.fi

Description

The routine ?lakf2 forms the $2*m*n$ by $2*m*n$ matrix Z .

$$Z = \begin{bmatrix} \text{kron}(In, A) & -\text{kron}(B^T, Im) \\ \text{kron}(In, D) & -\text{kron}(E^T, Im) \end{bmatrix},$$

where In is the identity matrix of size n and X^T is the transpose of X . $\text{kron}(X, Y)$ is the Kronecker product between the matrices X and Y .

Input Parameters

m	INTEGER. Size of matrix, $m \geq 1$
n	INTEGER. Size of matrix, $n \geq 1$
a	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size lda -by- n . The matrix A in the output matrix Z .
lda	INTEGER. The leading dimension of a , b , d , and e . $lda \geq m+n$.
b	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2, DOUBLE COMPLEX for zlakf2, Array, size lda by n . Matrix used in forming the output matrix Z .
d	REAL for slakf2, DOUBLE PRECISION for dlakf2, COMPLEX for clakf2,

DOUBLE COMPLEX for `zlakf2`,
 Array, size `lda` by `m`. Matrix used in forming the output matrix `Z`.

`e` REAL for `slakf2`,
 DOUBLE PRECISION for `dlakf2`,
 COMPLEX for `clakf2`,
 DOUBLE COMPLEX for `zlakf2`,
 Array, size `lda` by `n`. Matrix used in forming the output matrix `Z`.

`ldz` INTEGER. The leading dimension of `Z`. $ldz \geq 2 * m * n$.

Output Parameters

`z` REAL for `slakf2`,
 DOUBLE PRECISION for `dlakf2`,
 COMPLEX for `clakf2`,
 DOUBLE COMPLEX for `zlakf2`,
 Array, size `ldz-by-2*m*n`. The resultant Kronecker $m*n*2$ -by- $m*n*2$ matrix.

?laln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```
call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale,
xnorm, info )
```

```
call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale,
xnorm, info )
```

Include Files

- `mkl.fi`

Description

The routine solves a system of the form

$$(ca*A - w*D)*X = s*B, \text{ or } (ca*A^T - w*D)*X = s*B$$

with possible scaling (s) and perturbation of A .

A is an na -by- na real matrix, ca is a real scalar, D is an na -by- na real diagonal matrix, w is a real or complex value, and X and B are na -by-1 matrices: real if w is real, complex if w is complex. The parameter na may be 1 or 2.

If w is complex, X and B are represented as na -by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor s (≤ 1) so chosen that X can be computed without overflow. X is further scaled if necessary to assure that $\text{norm}(ca*A - w*D)*\text{norm}(X)$ is less than overflow.

If both singular values of $(ca*A - w*D)$ are less than $smin$, $smin*I$ (where I stands for identity) will be used instead of $(ca*A - w*D)$. If only one singular value is less than $smin$, one element of $(ca*A - w*D)$ will be perturbed enough to make the smallest singular value roughly $smin$.

If both singular values are at least $smin$, $(ca*A - w*D)$ will not be perturbed. In any case, the perturbation will be at most some small multiple of $\max(smin, ulp*norm(ca*A - w*D))$.

The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.

NOTE

All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

<i>trans</i>	LOGICAL. If <i>trans</i> = .TRUE., <i>A</i> - transpose will be used. If <i>trans</i> = .FALSE., <i>A</i> will be used (not transposed.)
<i>na</i>	INTEGER. The size of the matrix <i>A</i> , possible values 1 or 2.
<i>nw</i>	INTEGER. This parameter must be 1 if <i>w</i> is real, and 2 if <i>w</i> is complex. Possible values 1 or 2.
<i>smin</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The desired lower bound on the singular values of <i>A</i> . This should be a safe distance away from underflow or overflow, for example, between (<i>underflow/machine_precision</i>) and (<i>machine_precision * overflow</i>). (See <i>bignum</i> and <i>ulp</i>).
<i>ca</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The coefficient by which <i>A</i> is multiplied.
<i>a</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. Array, DIMENSION (<i>lda,na</i>). The <i>na</i> -by- <i>na</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> . Must be at least <i>na</i> .
<i>d1, d2</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The (1,1) and (2,2) elements in the diagonal matrix <i>D</i> , respectively. <i>d2</i> is not used if <i>nw</i> = 1.
<i>b</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2.

Array, DIMENSION (ldb, nw). The na -by- nw matrix B (right-hand side). If $nw = 2$ (w is complex), column 1 contains the real part of B and column 2 contains the imaginary part.

ldb INTEGER. The leading dimension of b . Must be at least na .

wr, wi REAL for `s1aln2`
DOUBLE PRECISION for `d1aln2`.

The real and imaginary part of the scalar w , respectively.
 wi is not used if $nw = 1$.

ldx INTEGER. The leading dimension of the output array x . Must be at least na .

Output Parameters

x REAL for `s1aln2`
DOUBLE PRECISION for `d1aln2`.

Array, DIMENSION (ldx, nw). The na -by- nw matrix X (unknowns), as computed by the routine. If $nw = 2$ (w is complex), on exit, column 1 will contain the real part of X and column 2 will contain the imaginary part.

scale REAL for `s1aln2`
DOUBLE PRECISION for `d1aln2`.

The scale factor that B must be multiplied by to insure that overflow does not occur when computing X . Thus $(ca^*A - w^*D) X$ will be $scale*B$, not B (ignoring perturbations of A .) It will be at most 1.

xnorm REAL for `s1aln2`
DOUBLE PRECISION for `d1aln2`.

The infinity-norm of X , when X is regarded as an na -by- nw real matrix.

info INTEGER.

An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if $(ca^*A - w^*D)$ had to be perturbed. The possible values are:

If $info = 0$: no error occurred, and $(ca^*A - w^*D)$ did not have to be perturbed.

If $info = 1$: $(ca^*A - w^*D)$ had to be perturbed to make its smallest (or only) singular value greater than $smim$.

NOTE

For higher speed, this routine does not check the inputs for errors.

slals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by `sgelsd`.

Syntax

```
call slals0(  icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
             ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )
```

```
call dlals0(  icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
             ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )
```

```
call clals0(  icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
             ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

```
call zlals0(  icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
             ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

Include Files

- `mkl.fi`

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix B in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is `givptr`; the pairs of columns/rows they were applied to are stored in `givcol`; and the c - and s -values of these rotations are stored in `givnum`.

(2L) Permutation. The $(nl+1)$ -st row of B is to be moved to the first row, and for $j=2:n$, `perm(j)`-th row of B is to be moved to the j -th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If `sqre = 1`, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

<code>icompq</code>	INTEGER. Specifies whether singular vectors are to be computed in factored form: If <code>icompq = 0</code> : Left singular vector matrix. If <code>icompq = 1</code> : Right singular vector matrix.
<code>nl</code>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<code>nr</code>	INTEGER. The row dimension of the lower block.

$nr \geq 1$.

sqre INTEGER.
 If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.
 If *sqre* = 1: the lower block is an *nr*-by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.

nrhs INTEGER. The number of columns of *B* and *bx*.
 Must be at least 1.

b REAL for slals0
 DOUBLE PRECISION for dlals0
 COMPLEX for clals0
 DOUBLE COMPLEX for zlals0.
 Array, DIMENSION (*ldb*, *nrhs*).
 Contains the right hand sides of the least squares problem in rows 1 through *m*.

ldb INTEGER. The leading dimension of *b*.
 Must be at least $\max(1, \max(m, n))$.

bx REAL for slals0
 DOUBLE PRECISION for dlals0
 COMPLEX for clals0
 DOUBLE COMPLEX for zlals0.
 Workspace array, DIMENSION (*ldb*, *nrhs*).

ldb INTEGER. The leading dimension of *bx*.

perm INTEGER. Array, DIMENSION (*n*).
 The permutations (from deflation and sorting) applied to the two blocks.

givptr INTEGER. The number of Givens rotations which took place in this subproblem.

givcol INTEGER. Array, DIMENSION (*ldgcol*, 2). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.

ldgcol INTEGER. The leading dimension of *givcol*, must be at least *n*.

givnum REAL for slals0/clals0
 DOUBLE PRECISION for dlals0/zlals0
 Array, DIMENSION (*ldgnum*, 2). Each number indicates the *c* or *s* value used in the corresponding Givens rotation.

<i>ldgnum</i>	INTEGER. The leading dimension of arrays <i>difr</i> , <i>poles</i> and <i>givnum</i> , must be at least <i>k</i> .
<i>poles</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>ldgnum</i> , 2). On entry, <i>poles</i> (1: <i>k</i> , 1) contains the new singular values obtained from solving the secular equation, and <i>poles</i> (1: <i>k</i> , 2) is an array containing the poles in the secular equation.
<i>difl</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>k</i>). On entry, <i>difl</i> (<i>i</i>) is the distance between <i>i</i> -th updated (undeflated) singular value and the <i>i</i> -th (undeflated) old singular value.
<i>difr</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>ldgnum</i> , 2). On entry, <i>difr</i> (<i>i</i> , 1) contains the distances between <i>i</i> -th updated (undeflated) singular value and the <i>i+1</i> -th (undeflated) old singular value. And <i>difr</i> (<i>i</i> , 2) is the normalizing factor for the <i>i</i> -th right singular vector.
<i>z</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Array, DIMENSION (<i>k</i>). Contains the components of the deflation-adjusted updating row vector.
<i>K</i>	INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.
<i>c</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if <i>sqre</i> = 0 and the <i>c</i> value of a Givens rotation related to the right null space if <i>sqre</i> = 1.
<i>s</i>	REAL for slals0/clals0 DOUBLE PRECISION for dlals0/zlals0 Contains garbage if <i>sqre</i> = 0 and the <i>s</i> value of a Givens rotation related to the right null space if <i>sqre</i> = 1.
<i>work</i>	REAL for slals0 DOUBLE PRECISION for dlals0 Workspace array, DIMENSION (<i>k</i>). Used with real flavors only.
<i>rwork</i>	REAL for clals0 DOUBLE PRECISION for zlals0

Workspace array, `DIMENSION` ($k*(1+nrhs) + 2*nrhs$). Used with complex flavors only.

Output Parameters

`b` On exit, contains the solution X in rows 1 through n .

`info` INTEGER.
 If `info` = 0: successful exit.
 If `info` = $-i < 0$, the i -th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.

Syntax

```
call slalsa( icompg, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call dlalsa( icompg, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call clalsa( icompg, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )
```

```
call zlalsa( icompg, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine is an intermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthogonal matrices.

If `icompg` = 0, ?lalsa applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if `icompg` = 1, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by ?lalsa.

Input Parameters

`icompg` INTEGER. Specifies whether the left or the right singular vector matrix is involved. If `icompg` = 0: left singular vector matrix is used
 If `icompg` = 1: right singular vector matrix is used.

`smlsiz` INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

`n` INTEGER. The row and column dimensions of the upper bidiagonal matrix.

<i>nrhs</i>	INTEGER. The number of columns of <i>b</i> and <i>bx</i> . Must be at least 1.
<i>b</i>	REAL for <i>slalsa</i> DOUBLE PRECISION for <i>dlalsa</i> COMPLEX for <i>clalsa</i> DOUBLE COMPLEX for <i>zlalsa</i> Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). Contains the right hand sides of the least squares problem in rows 1 through <i>m</i> .
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> in the calling subprogram. Must be at least $\max(1, \max(m, n))$.
<i>ldbx</i>	INTEGER. The leading dimension of the output array <i>bx</i> .
<i>u</i>	REAL for <i>slalsa/clalsa</i> DOUBLE PRECISION for <i>dlalsa/zlalsa</i> Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i>). On entry, <i>u</i> contains the left singular vector matrices of all subproblems at the bottom level.
<i>ldu</i>	INTEGER, $ldu \geq n$. The leading dimension of arrays <i>u</i> , <i>vt</i> , <i>difl</i> , <i>difr</i> , <i>poles</i> , <i>givnum</i> , and <i>z</i> .
<i>vt</i>	REAL for <i>slalsa/clalsa</i> DOUBLE PRECISION for <i>dlalsa/zlalsa</i> Array, DIMENSION(<i>ldu</i> , <i>smlsiz</i> + 1). On entry, <i>vt</i> ^T (for real flavors) or <i>vt</i> ^H (for complex flavors) contains the right singular vector matrices of all subproblems at the bottom level.
<i>k</i>	INTEGER array, DIMENSION (<i>n</i>).
<i>difl</i>	REAL for <i>slalsa/clalsa</i> DOUBLE PRECISION for <i>dlalsa/zlalsa</i> Array, DIMENSION (<i>ldu</i> , <i>nlvl</i>), where $nlvl = \text{int}(\log_2(n / (smlsiz+1))) + 1$.
<i>difr</i>	REAL for <i>slalsa/clalsa</i> DOUBLE PRECISION for <i>dlalsa/zlalsa</i> Array, DIMENSION(<i>ldu</i> , $2*nlvl$). On entry, <i>difl</i> (*, <i>i</i>) and <i>difr</i> (*, $2i - 1$) record distances between singular values on the <i>i</i> -th level and singular values on the (<i>i</i> - 1)-th level, and <i>difr</i> (*, $2i$) record the normalizing factors of the right singular vectors matrices of subproblems on <i>i</i> -th level.
<i>z</i>	REAL for <i>slalsa/clalsa</i> DOUBLE PRECISION for <i>dlalsa/zlalsa</i> Array, DIMENSION (<i>ldu</i> , <i>nlvl</i>). On entry, <i>z</i> (1, <i>i</i>) contains the components of the deflation- adjusted updating the row vector for subproblems on the <i>i</i> -th level.

<i>poles</i>	<p>REAL for <i>slalsa/clalsa</i></p> <p>DOUBLE PRECISION for <i>dlalsa/zlalsa</i></p> <p>Array, DIMENSION (<i>Idu</i>, $2*n_{lvl}$).</p> <p>On entry, <i>poles</i>(*, $2i-1: 2i$) contains the new and old singular values involved in the secular equations on the <i>i</i>-th level.</p>
<i>givptr</i>	<p>INTEGER. Array, DIMENSION (<i>n</i>).</p> <p>On entry, <i>givptr</i>(<i>i</i>) records the number of Givens rotations performed on the <i>i</i>-th problem on the computation tree.</p>
<i>givcol</i>	<p>INTEGER. Array, DIMENSION (<i>ldgcol</i>, $2*n_{lvl}$). On entry, for each <i>i</i>, <i>givcol</i>(*, $2i-1: 2i$) records the locations of Givens rotations performed on the <i>i</i>-th level on the computation tree.</p>
<i>ldgcol</i>	<p>INTEGER, $ldgcol \geq n$. The leading dimension of arrays <i>givcol</i> and <i>perm</i>.</p>
<i>perm</i>	<p>INTEGER. Array, DIMENSION (<i>ldgcol</i>, <i>n_{lvl}</i>). On entry, <i>perm</i>(*, <i>i</i>) records permutations done on the <i>i</i>-th level of the computation tree.</p>
<i>givnum</i>	<p>REAL for <i>slalsa/clalsa</i></p> <p>DOUBLE PRECISION for <i>dlalsa/zlalsa</i></p> <p>Array, DIMENSION (<i>Idu</i>, $2*n_{lvl}$). On entry, <i>givnum</i>(*, $2i-1 : 2i$) records the <i>c</i> and <i>s</i> values of Givens rotations performed on the <i>i</i>-th level on the computation tree.</p>
<i>c</i>	<p>REAL for <i>slalsa/clalsa</i></p> <p>DOUBLE PRECISION for <i>dlalsa/zlalsa</i></p> <p>Array, DIMENSION (<i>n</i>). On entry, if the <i>i</i>-th subproblem is not square, <i>c</i>(<i>i</i>) contains the <i>c</i> value of a Givens rotation related to the right null space of the <i>i</i>-th subproblem.</p>
<i>s</i>	<p>REAL for <i>slalsa/clalsa</i></p> <p>DOUBLE PRECISION for <i>dlalsa/zlalsa</i></p> <p>Array, DIMENSION (<i>n</i>). On entry, if the <i>i</i>-th subproblem is not square, <i>s</i>(<i>i</i>) contains the <i>s</i>-value of a Givens rotation related to the right null space of the <i>i</i>-th subproblem.</p>
<i>work</i>	<p>REAL for <i>slalsa</i></p> <p>DOUBLE PRECISION for <i>dlalsa</i></p> <p>Workspace array, DIMENSION at least (<i>n</i>). Used with real flavors only.</p>
<i>rwork</i>	<p>REAL for <i>clalsa</i></p> <p>DOUBLE PRECISION for <i>zlalsa</i></p> <p>Workspace array, DIMENSION at least $\max(n, (smlsz+1)*nrhs*3)$. Used with complex flavors only.</p>
<i>iwork</i>	<p>INTEGER.</p> <p>Workspace array, DIMENSION at least ($3n$).</p>

Output Parameters

<i>b</i>	On exit, contains the solution X in rows 1 through n .
<i>bx</i>	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa DOUBLE COMPLEX for zlalsa Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). On exit, the result of applying the left or right singular vector matrix to <i>b</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value.

?lalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call dlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call clalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
call zlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork, info )
```

Include Files

- mkl.fi

Description

The routine uses the singular value decomposition of A to solve the least squares problem of finding X to minimize the Euclidean norm of each column of $A*X-B$, where A is n -by- n upper bidiagonal, and X and B are n -by- $nrhs$. The solution X overwrites B .

The singular values of A smaller than *rcond* times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in *d* in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

<i>uplo</i>	CHARACTER*1. If <i>uplo</i> = 'U', <i>d</i> and <i>e</i> define an upper bidiagonal matrix.
-------------	--

If *uplo* = 'L', *d* and *e* define a lower bidiagonal matrix.

smlsiz INTEGER. The maximum size of the subproblems at the bottom of the computation tree.

n INTEGER. The dimension of the bidiagonal matrix.
 $n \geq 0$.

nrhs INTEGER. The number of columns of *B*. Must be at least 1.

d REAL for slalsd/clalsd
DOUBLE PRECISION for dlalsd/zlalsd
Array, DIMENSION (*n*). On entry, *d* contains the main diagonal of the bidiagonal matrix.

e REAL for slalsd/clalsd
DOUBLE PRECISION for dlalsd/zlalsd
Array, DIMENSION (*n*-1). Contains the super-diagonal entries of the bidiagonal matrix. On exit, *e* is destroyed.

b REAL for slalsd
DOUBLE PRECISION for dlalsd
COMPLEX for clalsd
DOUBLE COMPLEX for zlalsd
Array, DIMENSION (*ldb*,*nrhs*).
On input, *b* contains the right hand sides of the least squares problem. On output, *b* contains the solution *X*.

ldb INTEGER. The leading dimension of *b* in the calling subprogram. Must be at least $\max(1, n)$.

rcond REAL for slalsd/clalsd
DOUBLE PRECISION for dlalsd/zlalsd
The singular values of *A* less than or equal to *rcond* times the largest singular value are treated as zero in solving the least squares problem. If *rcond* is negative, machine precision is used instead. For example, for the least squares problem $\text{diag}(S) * X = B$, where $\text{diag}(S)$ is a diagonal matrix of singular values, the solution is $X(i) = B(i) / S(i)$ if $S(i)$ is greater than $rcond * \max(S)$, and $X(i) = 0$ if $S(i)$ is less than or equal to $rcond * \max(S)$.

rank INTEGER. The number of singular values of *A* greater than *rcond* times the largest singular value.

work REAL for slalsd
DOUBLE PRECISION for dlalsd
COMPLEX for clalsd
DOUBLE COMPLEX for zlalsd

Workspace array.

DIMENSION for real flavors at least

$$(9n+2n*smlsiz+8n*nlvl+n*nrhs+(smlsiz+1)^2),$$

where

$$nlvl = \max(0, \text{int}(\log_2(n/(smlsiz+1))) + 1).$$

DIMENSION for complex flavors is $(n*nrhs)$.

rwork

REAL for `clalsd`

DOUBLE PRECISION for `zlalsd`

Workspace array, used with complex flavors only.

DIMENSION at least $(9n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2)$,

where

$$nlvl = \max(0, \text{int}(\log_2(\min(m,n)/(smlsiz+1))) + 1).$$

iwork

INTEGER.

Workspace array of DIMENSION $(3n*nlvl + 11n)$.

Output Parameters

d

On exit, if *info* = 0, *d* contains singular values of the bidiagonal matrix.

e

On exit, destroyed.

b

On exit, *b* contains the solution *X*.

info

INTEGER.

If *info* = 0: successful exit.

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

If *info* > 0: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns *info*/(*n*+1) through *mod*(*info*,*n*+1).

?lamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.

Syntax

call slamrg(*n1*, *n2*, *a*, *strd1*, *strd2*, *index*)

call dlamrg(*n1*, *n2*, *a*, *strd1*, *strd2*, *index*)

Include Files

- `mkl.fi`

Description

The routine creates a permutation list which will merge the elements of a (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

$n1, n2$	INTEGER. These arguments contain the respective lengths of the two sorted lists to be merged.
a	REAL for <code>slamrg</code> DOUBLE PRECISION for <code>dlamrg</code> . Array, DIMENSION $(n1+n2)$. The first $n1$ elements of a contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final $n2$ elements.
$strd1, strd2$	INTEGER. These are the strides to be taken through the array a . Allowable strides are 1 and -1. They indicate whether a subset of a is sorted in ascending ($strdx = 1$) or descending ($strdx = -1$) order.

Output Parameters

$index$	INTEGER. Array, DIMENSION $(n1+n2)$. On exit, this array will contain a permutation such that if $b(i) = a(index(i))$ for $i=1, n1+n2$, then b will be sorted in ascending order.
---------	--

?laneg

Computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal T - $\sigma I = L^*D^*L^T$.

Syntax

```
value = slaneg( n, d, lld, sigma, pivmin, r )
```

```
value = dlaneg( n, d, lld, sigma, pivmin, r )
```

Include Files

- `mk1.fi`

Description

The routine computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal T - $\sigma I = L^*D^*L^T$. This implementation works directly on the factors without forming the tridiagonal matrix T . The Sturm count is also the number of eigenvalues of T less than σ . This routine is called from `?larb`. The current routine does not use the `pivmin` parameter but rather requires IEEE-754 propagation of infinities and NaNs (NaN stands for 'Not A Number'). This routine also has no input range restrictions but does require default exception handling such that $x/0$ produces `Inf` when x is non-zero, and `Inf/Inf` produces `NaN`. (For more information see [[Marques06](#)]).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix.
<i>d</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the matrix <i>D</i> .
<i>lld</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (<i>n</i> -1). Contains (<i>n</i> -1) elements $L(i)*L(i)*D(i)$.
<i>sigma</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Shift amount in $T-sigma*I = L*D*L**T$.
<i>pivmin</i>	REAL for slaneg DOUBLE PRECISION for dlaneg The minimum pivot in the Sturm sequence. May be used when zero pivots are encountered on non-IEEE-754 architectures.
<i>r</i>	INTEGER. The twist index for the twisted factorization that is used for the negcount.

Output Parameters

<i>value</i>	INTEGER. The number of negative pivots encountered while factoring.
--------------	---

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

val = slangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = dlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = clangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

val = zlangb(*norm*, *n*, *kl*, *ku*, *ab*, *ldab*, *work*)

Include Files

- mkl.fi

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n band matrix A , with kl sub-diagonals and ku super-diagonals.

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <p>= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A.</p> <p>= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),</p> <p>= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),</p> <p>= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).</p>
<i>n</i>	<p>INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, $?langb$ is set to zero.</p>
<i>kl</i>	<p>INTEGER. The number of sub-diagonals of the matrix A. $kl \geq 0$.</p>
<i>ku</i>	<p>INTEGER. The number of super-diagonals of the matrix A. $ku \geq 0$.</p>
<i>ab</i>	<p>REAL for slangb DOUBLE PRECISION for dlangb COMPLEX for clangb DOUBLE COMPLEX for zlangb</p> <p>Array, DIMENSION ($ldab, n$).</p> <p>The band matrix A, stored in rows 1 to $kl+ku+1$. The j-th column of A is stored in the j-th column of the array ab as follows:</p> $ab(ku+1+i-j, j) = a(i, j)$ <p>for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.</p>
<i>ldab</i>	<p>INTEGER. The leading dimension of the array ab.</p> <p>$ldab \geq kl+ku+1$.</p>
<i>work</i>	<p>REAL for slangb/clangb DOUBLE PRECISION for dlangb/zlangb</p> <p>Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when $norm = 'I'$; otherwise, $work$ is not referenced.</p>

Output Parameters

<i>val</i>	<p>REAL for slangb/clangb DOUBLE PRECISION for dlangb/zlangb</p> <p>Value returned by the function.</p>
------------	--

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange( norm, m, n, a, lda, work )
```

```
val = dlange( norm, m, n, a, lda, work )
```

```
val = clange( norm, m, n, a, lda, work )
```

```
val = zlange( norm, m, n, a, lda, work )
```

Include Files

- mkl.fi

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A.

Input Parameters

The data types are given for the Fortran interface.

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>m</i>	<p>INTEGER. The number of rows of the matrix A.</p> <p>$m \geq 0$. When $m = 0$, ?lange is set to zero.</p>
<i>n</i>	<p>INTEGER. The number of columns of the matrix A.</p> <p>$n \geq 0$. When $n = 0$, ?lange is set to zero.</p>
<i>a</i>	<p>REAL for slange</p> <p>DOUBLE PRECISION for dlange</p> <p>COMPLEX for clange</p> <p>DOUBLE COMPLEX for zlange</p> <p>Array, DIMENSION (<i>lda</i>,<i>n</i>).</p> <p>Array <i>a</i> contains the <i>m</i>-by-<i>n</i> matrix A.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>.</p>

work REAL for slange and clange.
 DOUBLE PRECISION for dlange and zlange.
 Workspace array, DIMENSION $\max(1, lwork)$, where $lwork \geq m$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slange/clange
 DOUBLE PRECISION for dlange/zlange
 Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

```
val = slangt( norm, n, dl, d, du )
val = dlangt( norm, n, dl, d, du )
val = clangt( norm, n, dl, d, du )
val = zlangt( norm, n, dl, d, du )
```

Include Files

- mkl.fi

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix *A*.

Input Parameters

norm CHARACTER*1. Specifies the value to be returned by the routine:
 = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix *A*.
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix *A* (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix *A* (maximum row sum),
 = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix *A* (square root of sum of squares).

n INTEGER. The order of the matrix *A*. $n \geq 0$. When $n = 0$, ?langt is set to zero.

dl, d, du REAL for slangt

DOUBLE PRECISION for dlangt

COMPLEX for clangt

DOUBLE COMPLEX for zlangt

Arrays: dl ($n-1$), d (n), du ($n-1$).

The array dl contains the ($n-1$) sub-diagonal elements of A .

The array d contains the diagonal elements of A .

The array du contains the ($n-1$) super-diagonal elements of A .

Output Parameters

val REAL for slangt/clangt
 DOUBLE PRECISION for dlangt/zlangt
 Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

val = slanh(*norm*, *n*, *a*, *lda*, *work*)

val = dlanh(*norm*, *n*, *a*, *lda*, *work*)

val = clanh(*norm*, *n*, *a*, *lda*, *work*)

val = zlanh(*norm*, *n*, *a*, *lda*, *work*)

Include Files

- mkl.fi

Description

The function ?lanhs returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix A .

The value *val* returned by the function is:

$val = \max(\text{abs}(A_{ij}))$, if *norm* = 'M' or 'm'

= $\text{norm}_1(A)$, if *norm* = '1' or 'O' or 'o'

= $\text{norm}_I(A)$, if *norm* = 'I' or 'i'

= $\text{norm}_F(A)$, if *norm* = 'F', 'f', 'E' or 'e'

where norm_1 denotes the 1-norm of a matrix (maximum column sum), norm_I denotes the infinity norm of a matrix (maximum row sum) and norm_F denotes the Frobenius norm of a matrix (square root of sum of squares). Note that $\max(\text{abs}(A_{ij}))$ is not a consistent matrix norm.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine as described above.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$. When $n = 0$, <i>?lanhs</i> is set to zero.
<i>a</i>	REAL for <i>slanhs</i> DOUBLE PRECISION for <i>dlanhs</i> COMPLEX for <i>clanhs</i> DOUBLE COMPLEX for <i>zlanhs</i> Array, DIMENSION (<i>lda</i> , <i>n</i>). The <i>n</i> -by- <i>n</i> upper Hessenberg matrix <i>A</i> ; the part of <i>A</i> below the first sub-diagonal is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(n, 1)$.
<i>work</i>	REAL for <i>slanhs</i> and <i>clanhs</i> . DOUBLE PRECISION for <i>dlanhs</i> and <i>zlanhs</i> . Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when <i>norm</i> = 'I'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for <i>slanhs</i> / <i>clanhs</i> DOUBLE PRECISION for <i>dlanhs</i> / <i>zlanhs</i> Value returned by the function.
------------	---

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```
val = slansb( norm, uplo, n, k, ab, ldab, work )
```

```
val = dlansb( norm, uplo, n, k, ab, ldab, work )
```

```
val = clansb( norm, uplo, n, k, ab, ldab, work )
```

```
val = zlansb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The function `?lansb` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n real/complex symmetric band matrix A , with k super-diagonals.

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the band matrix A is supplied. If <i>uplo</i> = 'U': upper triangular part is supplied; If <i>uplo</i> = 'L': lower triangular part is supplied.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A. $n \geq 0$.</p> <p>When $n = 0$, <code>?lansb</code> is set to zero.</p>
<i>k</i>	<p>INTEGER. The number of super-diagonals or sub-diagonals of the band matrix A. $k \geq 0$.</p>
<i>ab</i>	<p>REAL for <code>slansb</code></p> <p>DOUBLE PRECISION for <code>dlansb</code></p> <p>COMPLEX for <code>clansb</code></p> <p>DOUBLE COMPLEX for <code>zlansb</code></p> <p>Array, DIMENSION (<i>ldab</i>,<i>n</i>).</p> <p>The upper or lower triangle of the symmetric band matrix A, stored in the first $k+1$ rows of <i>ab</i>. The j-th column of A is stored in the j-th column of the array <i>ab</i> as follows:</p> <p>if <i>uplo</i> = 'U', $ab(k+1+i-j, j) = a(i, j)$</p> <p>for $\max(1, j-k) \leq i \leq j$;</p> <p>if <i>uplo</i> = 'L', $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$.</p>
<i>ldab</i>	<p>INTEGER. The leading dimension of the array <i>ab</i>.</p> <p>$ldab \geq k+1$.</p>
<i>work</i>	<p>REAL for <code>slansb</code> and <code>clansb</code>.</p> <p>DOUBLE PRECISION for <code>dlansb</code> and <code>zlansb</code>.</p> <p>Workspace array, DIMENSION ($\max(1, lwork)$), where</p> <p>$lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.</p>

Output Parameters

val REAL for slansb/clansb
 DOUBLE PRECISION for dlansb/zlansb
 Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

```
val = clanhb( norm, uplo, n, k, ab, ldab, work )
val = zlanhb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- mkl.fi

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n Hermitian band matrix A , with k super-diagonals.

Input Parameters

norm CHARACTER*1. Specifies the value to be returned by the routine:
 = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
 = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

uplo CHARACTER*1.
 Specifies whether the upper or lower triangular part of the band matrix A is supplied.
 If *uplo* = 'U': upper triangular part is supplied;
 If *uplo* = 'L': lower triangular part is supplied.

n INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lanhb is set to zero.

k INTEGER. The number of super-diagonals or sub-diagonals of the band matrix A .
 $k \geq 0$.

ab COMPLEX for clanhb.

DOUBLE COMPLEX for zlanhb.

Array, DIMENSION ($ldaB, n$). The upper or lower triangle of the Hermitian band matrix A , stored in the first $k+1$ rows of ab . The j -th column of A is stored in the j -th column of the array ab as follows:

if $uplo = 'U'$, $ab(k+1+i-j, j) = a(i, j)$

for $\max(1, j-k) \leq i \leq j$;

if $uplo = 'L'$, $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$.

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

ldab

INTEGER. The leading dimension of the array ab . $ldab \geq k+1$.

work

REAL for clanhb.

DOUBLE PRECISION for zlanhb.

Workspace array, DIMENSION $\max(1, lwork)$, where

$lwork \geq n$ when $norm = 'I'$ or $'1'$ or $'O'$; otherwise, $work$ is not referenced.

Output Parameters

val

REAL for slanhb/clanhb

DOUBLE PRECISION for dlanhb/zlanhb

Value returned by the function.

?lansp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

$val = slansp(norm, uplo, n, ap, work)$

$val = dlansp(norm, uplo, n, ap, work)$

$val = clansp(norm, uplo, n, ap, work)$

$val = zlansp(norm, uplo, n, ap, work)$

Include Files

- mkl.fi

Description

The function ?lansp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A , supplied in packed form.

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <p>= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A.</p> <p>= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),</p> <p>= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),</p> <p>= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).</p>
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the symmetric matrix A is supplied.</p> <p>If <i>uplo</i> = 'U': Upper triangular part of A is supplied</p> <p>If <i>uplo</i> = 'L': Lower triangular part of A is supplied.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <i>?lansp</i> is set to zero.</p>
<i>ap</i>	<p>REAL for <i>slansp</i></p> <p>DOUBLE PRECISION for <i>dlansp</i></p> <p>COMPLEX for <i>clansp</i></p> <p>DOUBLE COMPLEX for <i>zlansp</i></p> <p>Array, DIMENSION $(n(n+1)/2)$.</p> <p>The upper or lower triangle of the symmetric matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array <i>ap</i> as follows:</p> <p>if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$;</p> <p>if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.</p>
<i>work</i>	<p>REAL for <i>slansp</i> and <i>clansp</i>.</p> <p>DOUBLE PRECISION for <i>dlansp</i> and <i>zlansp</i>.</p> <p>Workspace array, DIMENSION $(\max(1, lwork))$, where $lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.</p>

Output Parameters

<i>val</i>	<p>REAL for <i>slansp/clansp</i></p> <p>DOUBLE PRECISION for <i>dlansp/zlansp</i></p> <p>Value returned by the function.</p>
------------	--

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

```
val = clanhp( norm, uplo, n, ap, work )
```

```
val = zlanhp( norm, uplo, n, ap, work )
```

Include Files

- mkl.fi

Description

The function ?lanhp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A , supplied in packed form.

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the Hermitian matrix A is supplied.</p> <p>If <i>uplo</i> = 'U': Upper triangular part of A is supplied</p> <p>If <i>uplo</i> = 'L': Lower triangular part of A is supplied.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A.</p> <p>$n \geq 0$. When $n = 0$, ?lanhp is set to zero.</p>
<i>ap</i>	<p>COMPLEX for clanhp.</p> <p>DOUBLE COMPLEX for zlanhp.</p> <p>Array, DIMENSION $(n(n+1)/2)$. The upper or lower triangle of the Hermitian matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array <i>ap</i> as follows:</p> <ul style="list-style-type: none"> if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$; if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.
<i>work</i>	<p>REAL for clanhp.</p>

DOUBLE PRECISION for `zlanhp`.

Workspace array, `DIMENSION(max(1, lwork))`, where

$lwork \geq n$ when `norm = 'I' or '1' or 'O'`; otherwise, `work` is not referenced.

Output Parameters

`val` REAL for `clanhp`.
DOUBLE PRECISION for `zlanhp`.
Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

```
val = slanst( norm, n, d, e )
val = dlanst( norm, n, d, e )
val = clanht( norm, n, d, e )
val = zlanht( norm, n, d, e )
```

Include Files

- `mk1.fi`

Description

The functions `?lanst/?lanht` return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix A .

Input Parameters

`norm` CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

`n` INTEGER. The order of the matrix A .
 $n \geq 0$. When $n = 0$, `?lanst/?lanht` is set to zero.

`d` REAL for `slanst/clanht`

DOUBLE PRECISION for dlanst/zlanht

Array, DIMENSION (n). The diagonal elements of A .

e

REAL for slanst

DOUBLE PRECISION for dlanst

COMPLEX for clanht

DOUBLE COMPLEX for zlanht

Array, DIMENSION ($n-1$).

The ($n-1$) sub-diagonal or super-diagonal elements of A .

Output Parameters

val

REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht

Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

$val = slansy(norm, uplo, n, a, lda, work)$

$val = dlansy(norm, uplo, n, a, lda, work)$

$val = clansy(norm, uplo, n, a, lda, work)$

$val = zlansy(norm, uplo, n, a, lda, work)$

Include Files

- mkl.fi

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A .

Input Parameters

The data types are given for the Fortran interface.

$norm$

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .

= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),

= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),

	= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced. = 'U': Upper triangular part of A is referenced. = 'L': Lower triangular part of A is referenced
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, <i>?lansy</i> is set to zero.
<i>a</i>	REAL for slansy DOUBLE PRECISION for dlansy COMPLEX for clansy DOUBLE COMPLEX for zlansy Array, size (lda, n). The symmetric matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(n, 1)$.
<i>work</i>	REAL for slansy and clansy. DOUBLE PRECISION for dlansy and zlansy. Workspace array, DIMENSION($\max(1, lwork)$), where $lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for slansy/clansy DOUBLE PRECISION for dlansy/zlansy Value returned by the function.
------------	---

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

$val = \text{clanhe}(norm, uplo, n, a, lda, work)$

```
val = zlanhe( norm, uplo, n, a, lda, work )
```

Include Files

- mkl.fi

Description

The function `?lanhe` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A .

Input Parameters

The data types are given for the Fortran interface.

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the Hermitian matrix A is to be referenced.</p> <ul style="list-style-type: none"> = 'U': Upper triangular part of A is referenced. = 'L': Lower triangular part of A is referenced
<i>n</i>	<p>INTEGER. The order of the matrix A. $n \geq 0$. When $n = 0$, <code>?lanhe</code> is set to zero.</p>
<i>a</i>	<p>COMPLEX for <code>clanhe</code>.</p> <p>DOUBLE COMPLEX for <code>zlanhe</code>.</p> <p>Array, size (lda, n). The Hermitian matrix A.</p> <p>If <code>uplo = 'U'</code>, the leading n-by-n upper triangular part of a contains the upper triangular part of the matrix A, and the strictly lower triangular part of a is not referenced.</p> <p>If <code>uplo = 'L'</code>, the leading n-by-n lower triangular part of a contains the lower triangular part of the matrix A, and the strictly upper triangular part of a is not referenced.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array a.</p> <p>$lda \geq \max(n, 1)$.</p>
<i>work</i>	<p>REAL for <code>clanhe</code>.</p> <p>DOUBLE PRECISION for <code>zlanhe</code>.</p> <p>Workspace array, DIMENSION $(\max(1, lwork))$, where</p>

$lwork \geq n$ when $norm = 'I'$ or $'1'$ or $'O'$; otherwise, $work$ is not referenced.

Output Parameters

val REAL for `clanhe`.
DOUBLE PRECISION for `zlanhe`.
Value returned by the function.

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb( norm, uplo, diag, n, k, ab, ldab, work )
```

Include Files

- `mkl.fi`

Description

The function `?lantb` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n triangular band matrix A , with $(k + 1)$ diagonals.

Input Parameters

$norm$ CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

$uplo$ CHARACTER*1.
Specifies whether the matrix A is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular.

$diag$ CHARACTER*1.

Specifies whether or not the matrix A is unit triangular.

= 'N': Non-unit triangular

= 'U': Unit triangular.

n INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, *?lantb* is set to zero.

k INTEGER. The number of super-diagonals of the matrix A if *uplo* = 'U', or the number of sub-diagonals of the matrix A if *uplo* = 'L'. $k \geq 0$.

ab REAL for *slantb*
 DOUBLE PRECISION for *dlantb*
 COMPLEX for *clantb*
 DOUBLE COMPLEX for *zlantb*

Array, DIMENSION (*ldab*,*n*). The upper or lower triangular band matrix A , stored in the first $k+1$ rows of *ab*.

The j -th column of A is stored in the j -th column of the array *ab* as follows:

if *uplo* = 'U', $ab(k+1+i-j, j) = a(i, j)$ for $\max(1, j-k) \leq i \leq j$;

if *uplo* = 'L', $ab(1+i-j, j) = a(i, j)$ for $j \leq i \leq \min(n, j+k)$.

Note that when *diag* = 'U', the elements of the array *ab* corresponding to the diagonal elements of the matrix A are not referenced, but are assumed to be one.

ldab INTEGER. The leading dimension of the array *ab*.
ldab $\geq k+1$.

work REAL for *slantb* and *clantb*.
 DOUBLE PRECISION for *dlantb* and *zlantb*.
 Workspace array, DIMENSION ($\max(1, lwork)$), where
lwork $\geq n$ when *norm* = 'I' ; otherwise, *work* is not referenced.

Output Parameters

val REAL for *slantb*/*clantb*.
 DOUBLE PRECISION for *dlantb*/*zlantb*.
 Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

val = *slantp*(*norm*, *uplo*, *diag*, *n*, *ap*, *work*)

```
val = dlantp( norm, uplo, diag, n, ap, work )
```

```
val = clantp( norm, uplo, diag, n, ap, work )
```

```
val = zlantp( norm, uplo, diag, n, ap, work )
```

Include Files

- mkl.fi

Description

The function ?lantp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix A , supplied in packed form.

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <p>= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A.</p> <p>= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),</p> <p>= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),</p> <p>= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).</p>
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the matrix A is upper or lower triangular.</p> <p>= 'U': Upper triangular</p> <p>= 'L': Lower triangular.</p>
<i>diag</i>	<p>CHARACTER*1.</p> <p>Specifies whether or not the matrix A is unit triangular.</p> <p>= 'N': Non-unit triangular</p> <p>= 'U': Unit triangular.</p>
<i>n</i>	<p>INTEGER. The order of the matrix A.</p> <p>$n \geq 0$. When $n = 0$, ?lantp is set to zero.</p>
<i>ap</i>	<p>REAL for slantp</p> <p>DOUBLE PRECISION for dlantp</p> <p>COMPLEX for clantp</p> <p>DOUBLE COMPLEX for zlantp</p> <p>Array, DIMENSION $(n(n+1)/2)$.</p> <p>The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array ap as follows:</p> <p>if $uplo = 'U'$, $ap(i + (j-1)j/2) = a(i, j)$ for $1 \leq i \leq j$;</p> <p>if $uplo = 'L'$, $ap(i + (j-1)(2n-j)/2) = a(i, j)$ for $j \leq i \leq n$.</p>

Note that when $diag = 'U'$, the elements of the array ap corresponding to the diagonal elements of the matrix A are not referenced, but are assumed to be one.

work

REAL for slantp and clantp.

DOUBLE PRECISION for dlantp and zlantp.

Workspace array, DIMENSION(max(1, $lwork$)), where $lwork \geq n$ when $norm = 'I'$; otherwise, $work$ is not referenced.

Output Parameters

val

REAL for slantp/clantp.

DOUBLE PRECISION for dlantp/zlantp.

Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

$val = slantr(norm, uplo, diag, m, n, a, lda, work)$

$val = dlantr(norm, uplo, diag, m, n, a, lda, work)$

$val = clantr(norm, uplo, diag, m, n, a, lda, work)$

$val = zlantr(norm, uplo, diag, m, n, a, lda, work)$

Include Files

- mkl.fi

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix A .

Input Parameters

The data types are given for the Fortran interface.

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .

= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),

= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),

= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the matrix <i>A</i> is upper or lower trapezoidal.</p> <p>= 'U': Upper trapezoidal</p> <p>= 'L': Lower trapezoidal.</p> <p>Note that <i>A</i> is triangular instead of trapezoidal if $m = n$.</p>
<i>diag</i>	<p>CHARACTER*1.</p> <p>Specifies whether or not the matrix <i>A</i> has unit diagonal.</p> <p>= 'N': Non-unit diagonal</p> <p>= 'U': Unit diagonal.</p>
<i>m</i>	<p>INTEGER. The number of rows of the matrix <i>A</i>. $m \geq 0$, and if <i>uplo</i> = 'U', $m \leq n$.</p> <p>When $m = 0$, <i>?lantr</i> is set to zero.</p>
<i>n</i>	<p>INTEGER. The number of columns of the matrix <i>A</i>. $n \geq 0$, and if <i>uplo</i> = 'L', $n \leq m$.</p> <p>When $n = 0$, <i>?lantr</i> is set to zero.</p>
<i>a</i>	<p>REAL for <i>slantr</i></p> <p>DOUBLE PRECISION for <i>dlantr</i></p> <p>COMPLEX for <i>clantr</i></p> <p>DOUBLE COMPLEX for <i>zlantr</i></p> <p>Array, DIMENSION (<i>lda</i>,<i>n</i>).</p> <p>The trapezoidal matrix <i>A</i> (<i>A</i> is triangular if $m = n$).</p> <p>If <i>uplo</i> = 'U', the leading <i>m</i>-by-<i>n</i> upper trapezoidal part of the array <i>a</i> contains the upper trapezoidal matrix, and the strictly lower triangular part of <i>A</i> is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>m</i>-by-<i>n</i> lower trapezoidal part of the array <i>a</i> contains the lower trapezoidal matrix, and the strictly upper triangular part of <i>A</i> is not referenced. Note that when <i>diag</i> = 'U', the diagonal elements of <i>A</i> are not referenced and are assumed to be one.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>.</p> <p>$lda \geq \max(m, 1)$.</p>
<i>work</i>	<p>REAL for <i>slantr/clantrp</i>.</p> <p>DOUBLE PRECISION for <i>dlantr/zlantr</i>.</p> <p>Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq m$ when <i>norm</i> = 'I'; otherwise, <i>work</i> is not referenced.</p>

Output Parameters

val REAL for *slantr/clantrp*.

DOUBLE PRECISION for dlantr/zlantr.

Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```
call slanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

```
call dlanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
```

Include Files

- mkl.fi

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

1. $cc = 0$ so that aa and dd are real eigenvalues of the matrix, or
2. $aa = dd$ and $bb*cc < 0$, so that $aa \pm \sqrt{bb*cc}$ are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that $\text{abs}(rt1r) \geq \text{abs}(rt2r)$.

Input Parameters

a, b, c, d REAL for slanv2
DOUBLE PRECISION for dlanv2.
On entry, elements of the input matrix.

Output Parameters

a, b, c, d On exit, overwritten by the elements of the standardized Schur form.

$rt1r, rt1i, rt2r, rt2i$ REAL for slanv2
DOUBLE PRECISION for dlanv2.
The real and imaginary parts of the eigenvalues.
If the eigenvalues are a complex conjugate pair, $rt1i > 0$.

cs, sn REAL for slanv2
DOUBLE PRECISION for dlanv2.

Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll( n, x, incx, Y, incy, ssmín )
call dlapll( n, x, incx, Y, incy, ssmín )
call clapll( n, x, incx, Y, incy, ssmín )
call zlapll( n, x, incx, Y, incy, ssmín )
```

Include Files

- mkl.fi

Description

Given two column vectors x and y of length n , let

$A = (xy)$ be the n -by-2 matrix.

The routine ?lapll first computes the QR factorization of A as $A = Q^*R$ and then computes the SVD of the 2-by-2 upper triangular matrix R . The smaller singular value of R is returned in $ssmin$, which is used as the measurement of the linear dependency of the vectors x and y .

Input Parameters

n	INTEGER. The length of the vectors x and y .
x	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION $(1+(n-1)incx)$. On entry, x contains the n -vector x .
y	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION $(1+(n-1)incy)$. On entry, y contains the n -vector y .
$incx$	INTEGER. The increment between successive elements of x ; $incx > 0$.
$incy$	INTEGER. The increment between successive elements of y ; $incy > 0$.

Output Parameters

<i>x</i>	On exit, <i>x</i> is overwritten.
<i>y</i>	On exit, <i>y</i> is overwritten.
<i>ssmin</i>	REAL for <code>slapll/clapll</code> DOUBLE PRECISION for <code>dlapll/zlapll</code> The smallest singular value of the <i>n</i> -by-2 matrix $A = (xy)$.

?lapmr

Rearranges rows of a matrix as specified by a permutation vector.

Syntax

```
call slapmr( forwrđ, m, n, x, ldx, k )
call dlapmr( forwrđ, m, n, x, ldx, k )
call clapmr( forwrđ, m, n, x, ldx, k )
call zlapmr( forwrđ, m, n, x, ldx, k )
call lapmr( x, k[, forwrđ] )
```

Include Files

- `mkl.fi`

Description

The ?lapmr routine rearranges the rows of the *m*-by-*n* matrix *X* as specified by the permutation $k(1), k(2), \dots, k(m)$ of the integers $1, \dots, m$.

If *forwrđ* = `.TRUE.`, forward permutation:

$X(k(i, *))$ is moved to $X(i, *)$ for $i = 1, 2, \dots, m$.

If *forwrđ* = `.FALSE.`, backward permutation:

$X(i, *)$ is moved to $X(k(i, *))$ for $i = 1, 2, \dots, m$.

Input Parameters

The data types are given for the Fortran interface.

<i>forwrđ</i>	LOGICAL. If <i>forwrđ</i> = <code>.TRUE.</code> , forward permutation. If <i>forwrđ</i> = <code>.FALSE.</code> , backward permutation.
<i>m</i>	INTEGER. The number of rows of the matrix <i>X</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>X</i> . $n \geq 0$.
<i>x</i>	REAL for <code>slapmr</code> DOUBLE PRECISION for <code>dlapmr</code>

COMPLEX for `clapmr`

DOUBLE COMPLEX for `zlapmr`

Array, size (ldx, n) On entry, the m -by- n matrix X .

ldx INTEGER. The leading dimension of the array X , $ldx \geq \max(1, m)$.

k INTEGER. Array, size (m) . On entry, k contains the permutation vector and is used as internal workspace.

Output Parameters

x On exit, x contains the permuted matrix X .

k On exit, k is reset to its original value.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `?lapmr` interface are as follows:

x Holds the matrix X of size (n, n) .

k Holds the vector of length m .

forwr Specifies the permutation. Must be `' .TRUE. '` or `' .FALSE. '`.

See Also

`?lapmt`

`?lapmt`

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt( forwr, m, n, x, ldx, k )
```

```
call dlapmt( forwr, m, n, x, ldx, k )
```

```
call clapmt( forwr, m, n, x, ldx, k )
```

```
call zlapmt( forwr, m, n, x, ldx, k )
```

Include Files

- `mkl.fi`

Description

The routine `?lapmt` rearranges the columns of the m -by- n matrix X as specified by the permutation $k(1), k(2), \dots, k(n)$ of the integers $1, \dots, n$.

If `forwr = .TRUE.`, forward permutation:

$X(*, k(j))$ is moved to $X(*, j)$ for $j=1, 2, \dots, n$.

If `forwr = .FALSE.`, backward permutation:

$X(*, j)$ is moved to $X(*, k(j))$ for $j = 1, 2, \dots, n$.

Input Parameters

<i>forwrd</i>	LOGICAL. If <i>forwrd</i> = .TRUE., forward permutation If <i>forwrd</i> = .FALSE., backward permutation
<i>m</i>	INTEGER. The number of rows of the matrix <i>X</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>X</i> . $n \geq 0$.
<i>x</i>	REAL for <i>slapmt</i> DOUBLE PRECISION for <i>dlapmt</i> COMPLEX for <i>clapmt</i> DOUBLE COMPLEX for <i>zlapmt</i> Array, DIMENSION (<i>ldx</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>X</i> .
<i>ldx</i>	INTEGER. The leading dimension of the array <i>X</i> , $ldx \geq \max(1, m)$.
<i>k</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, <i>k</i> contains the permutation vector and is used as internal workspace.

Output Parameters

<i>x</i>	On exit, <i>x</i> contains the permuted matrix <i>X</i> .
<i>k</i>	On exit, <i>k</i> is reset to its original value.

See Also

?lapmr

?lapy2

Returns $\sqrt{x^2+y^2}$.

Syntax

val = slapy2(*x*, *y*)

val = dlapy2(*x*, *y*)

Include Files

- mkl.fi

Description

The function ?lapy2 returns $\sqrt{x^2+y^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

<i>x</i> , <i>y</i>	REAL for slapy2
---------------------	-----------------

DOUBLE PRECISION for dlapy2

Specify the input values x and y .

Output Parameters

val REAL for slapy2
 DOUBLE PRECISION for dlapy2.
 Value returned by the function.
 If *val* = -1D0, the first argument was NaN.
 If *val* = -2D0, the second argument was NaN.

?lapy3

Returns $\text{sqrt}(x^2+y^2+z^2)$.

Syntax

val = slapy3(*x*, *y*, *z*)

val = dlapy3(*x*, *y*, *z*)

Include Files

- mkl.fi

Description

The function ?lapy3 returns $\text{sqrt}(x^2+y^2+z^2)$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

x, *y*, *z* REAL for slapy3
 DOUBLE PRECISION for dlapy3
 Specify the input values x , y and z .

Output Parameters

val REAL for slapy3
 DOUBLE PRECISION for dlapy3.
 Value returned by the function.
 If *val* = -1D0, the first argument was NaN.
 If *val* = -2D0, the second argument was NaN.
 If *val* = -3D0, the third argument was NaN.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by ?gbequ.

Syntax

```
call slaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call dlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call claqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call zlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a general m -by- n band matrix A with kl subdiagonals and ku superdiagonals using the row and column scaling factors in the vectors r and c .

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A . $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A . $ku \geq 0$.
ab	REAL for slaqgb DOUBLE PRECISION for dlaqgb COMPLEX for claqgb DOUBLE COMPLEX for zlaqgb Array, DIMENSION ($ldab, n$). On entry, the matrix A in band storage, in rows 1 to $kl+ku+1$. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.
$ldab$	INTEGER. The leading dimension of the array ab . $lda \geq kl+ku+1$.
$amax$	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Absolute value of largest matrix entry.
r, c	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Arrays $r(m)$, $c(n)$. Contain the row and column scale factors for A , respectively.
$rowcnd$	REAL for slaqgb/claqgb DOUBLE PRECISION for dlaqgb/zlaqgb Ratio of the smallest $r(i)$ to the largest $r(i)$.

colcnd REAL for slaqgb/claqgb
 DOUBLE PRECISION for dlaqgb/zlaqgb
 Ratio of the smallest $c(i)$ to the largest $c(i)$.

Output Parameters

ab On exit, the equilibrated matrix, in the same storage format as A .
 See *equed* for the form of the equilibrated matrix.

equed CHARACTER*1.
 Specifies the form of equilibration that was done.
 If *equed* = 'N': No equilibration
 If *equed* = 'R': Row equilibration, that is, A has been premultiplied by $\text{diag}(r)$.
 If *equed* = 'C': Column equilibration, that is, A has been postmultiplied by $\text{diag}(c)$.
 If *equed* = 'B': Both row and column equilibration, that is, A has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If $\text{rowcnd} < \text{thresh}$, row scaling is done, and if $\text{colcnd} < \text{thresh}$, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.

Syntax

```
call slaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call zlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a general m -by- n matrix A using the row and column scaling factors in the vectors r and c .

Input Parameters

m INTEGER. The number of rows of the matrix A .

	$m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	REAL for <i>slaqge</i> DOUBLE PRECISION for <i>dlaqge</i> COMPLEX for <i>claqge</i> DOUBLE COMPLEX for <i>zlaqge</i> Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(m, 1)$.
<i>r</i>	REAL for <i>slanqge/claqge</i> DOUBLE PRECISION for <i>dlaqge/zlaqge</i> Array, DIMENSION (<i>m</i>). The row scale factors for <i>A</i> .
<i>c</i>	REAL for <i>slanqge/claqge</i> DOUBLE PRECISION for <i>dlaqge/zlaqge</i> Array, DIMENSION (<i>n</i>). The column scale factors for <i>A</i> .
<i>rowcnd</i>	REAL for <i>slanqge/claqge</i> DOUBLE PRECISION for <i>dlaqge/zlaqge</i> Ratio of the smallest <i>r</i> (<i>i</i>) to the largest <i>r</i> (<i>i</i>).
<i>colcnd</i>	REAL for <i>slanqge/claqge</i> DOUBLE PRECISION for <i>dlaqge/zlaqge</i> Ratio of the smallest <i>c</i> (<i>i</i>) to the largest <i>c</i> (<i>i</i>).
<i>amax</i>	REAL for <i>slanqge/claqge</i> DOUBLE PRECISION for <i>dlaqge/zlaqge</i> Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	On exit, the equilibrated matrix. See <i>equed</i> for the form of the equilibrated matrix.
<i>equed</i>	CHARACTER*1. Specifies the form of equilibration that was done. If <i>equed</i> = 'N': No equilibration If <i>equed</i> = 'R': Row equilibration, that is, <i>A</i> has been premultiplied by <i>diag</i> (<i>r</i>).

If *equed* = 'C': Column equilibration, that is, *A* has been postmultiplied by $\text{diag}(c)$.

If *equed* = 'B': Both row and column equilibration, that is, *A* has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If $\text{rowcnd} < \text{thresh}$, row scaling is done, and if $\text{colcnd} < \text{thresh}$, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, row scaling is done.

?laqhb

Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call claqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

```
call zlaqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a Hermetian band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix <i>A</i> if <i>uplo</i> = 'U', or the number of sub-diagonals if <i>uplo</i> = 'L'. $kd \geq 0$.
<i>ab</i>	COMPLEX for claqhb DOUBLE COMPLEX for zlaqhb Array, DIMENSION (<i>ldab</i> , <i>n</i>). On entry, the upper or lower triangle of the band matrix <i>A</i> , stored in the first <i>kd</i> +1 rows of the array. The <i>j</i> -th column of <i>A</i> is stored in the <i>j</i> -th column of the array <i>ab</i> as follows:

if *uplo* = 'U', $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$;
 if *uplo* = 'L', $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab INTEGER. The leading dimension of the array *ab*.
ldab $\geq kd+1$.

scond REAL for *claqsb*
 DOUBLE PRECISION for *zlaqsb*
 Ratio of the smallest $s(i)$ to the largest $s(i)$.

amax REAL for *claqsb*
 DOUBLE PRECISION for *zlaqsb*
 Absolute value of largest matrix entry.

Output Parameters

ab On exit, if *info* = 0, the triangular factor *U* or *L* from the Cholesky factorization $A = U^H * U$ or $A = L * L^H$ of the band matrix *A*, in the same storage format as *A*.

s REAL for *claqsb*
 DOUBLE PRECISION for *zlaqsb*
 Array, DIMENSION (*n*). The scale factors for *A*.

equed CHARACTER*1.
 Specifies whether or not equilibration was done.
 If *equed* = 'N': No equilibration.
 If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done.

The values *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call claqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Include Files

- `mk1.fi`

Description

The routine computes a *QR* factorization with column pivoting of the block $A(\text{offset}+1:m, 1:n)$. The block $A(1:\text{offset}, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>offset</i>	INTEGER. The number of rows of the matrix <i>A</i> that must be pivoted but not factorized. $\text{offset} \geq 0$.
<i>a</i>	REAL for <code>slaqp2</code> DOUBLE PRECISION for <code>dlaqp2</code> COMPLEX for <code>claqp2</code> DOUBLE COMPLEX for <code>zlaqp2</code> Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $\text{lda} \geq \max(1, m)$.
<i>jpvt</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, if $\text{jpvt}(i) \neq 0$, the <i>i</i> -th column of <i>A</i> is permuted to the front of $A*P$ (a leading column); if $\text{jpvt}(i) = 0$, the <i>i</i> -th column of <i>A</i> is a free column.
<i>vn1, vn2</i>	REAL for <code>slaqp2/claqp2</code> DOUBLE PRECISION for <code>dlaqp2/zlaqp2</code> Arrays, DIMENSION (<i>n</i>) each. Contain the vectors with the partial and exact column norms, respectively.
<i>work</i>	REAL for <code>slaqp2</code> DOUBLE PRECISION for <code>dlaqp2</code> COMPLEX for <code>claqp2</code> DOUBLE COMPLEX for <code>zlaqp2</code> Workspace array, DIMENSION (<i>n</i>).

Output Parameters

<i>a</i>	On exit, the upper triangle of block $A(\text{offset}+1:m, 1:n)$ is the triangular factor obtained; the elements in block $A(\text{offset}+1:m, 1:n)$ below the diagonal, together with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors. Block $A(1:\text{offset}, 1:n)$ has been accordingly pivoted, but not factorized.
----------	--

<i>jpvt</i>	On exit, if $jpvt(i) = k$, then the i -th column of A^*P was the k -th column of A .
<i>tau</i>	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 DOUBLE COMPLEX for zlaqp2 Array, DIMENSION($\min(m, n)$). The scalar factors of the elementary reflectors.
<i>vn1, vn2</i>	Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3.

Syntax

```
call slaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call dlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call claqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call zlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
```

Include Files

- mkl.fi

Description

The routine computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3. The routine tries to factorize NB columns from A starting from the row $offset+1$, and updates all of the matrix with BLAS level 3 routine ?gemm.

In some cases, due to catastrophic cancellations, ?laqps cannot factorize NB columns. Hence, the actual number of factorized columns is returned in kb .

Block $A(1:offset, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>offset</i>	INTEGER. The number of rows of A that have been factorized in previous steps.
<i>nb</i>	INTEGER. The number of columns to factorize.
<i>a</i>	REAL for slaqps DOUBLE PRECISION for dlaqps

COMPLEX for claqps
DOUBLE COMPLEX for zlaqps
Array, DIMENSION (lda, n).
On entry, the m -by- n matrix A .

lda INTEGER. The leading dimension of the array a .
 $lda \geq \max(1, m)$.

$jpvt$ INTEGER. Array, DIMENSION (n).
If $jpvt(I) = k$ then column k of the full matrix A has been permuted into position i in AP.

$vn1, vn2$ REAL for slaqps/claqps
DOUBLE PRECISION for dlaqps/zlaqps
Arrays, DIMENSION (n) each. Contain the vectors with the partial and exact column norms, respectively.

$auxv$ REAL for slaqps
DOUBLE PRECISION for dlaqps
COMPLEX for claqps
DOUBLE COMPLEX for zlaqps
Array, DIMENSION (nb). Auxiliary vector.

f REAL for slaqps
DOUBLE PRECISION for dlaqps
COMPLEX for claqps
DOUBLE COMPLEX for zlaqps
Array, DIMENSION (ldf, nb). For real flavors, matrix $F^T = L*Y^T*A$. For complex flavors, matrix $F^H = L*Y^H*A$.

ldf INTEGER. The leading dimension of the array f .
 $ldf \geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns actually factorized.

a On exit, block $A(offset+1:m, 1:kb)$ is the triangular factor obtained and block $A(1:offset, 1:n)$ has been accordingly pivoted, but not factorized. The rest of the matrix, block $A(offset+1:m, kb+1:n)$ has been updated.

$jpvt$ INTEGER array, DIMENSION (n). If $jpvt(I) = k$ then column k of the full matrix A has been permuted into position i in AP.

tau REAL for slaqps
DOUBLE PRECISION for dlaqps

	COMPLEX for <code>claqps</code>
	DOUBLE COMPLEX for <code>zlaqps</code>
	Array, DIMENSION (<i>kb</i>). The scalar factors of the elementary reflectors.
<i>vn1, vn2</i>	The vectors with the partial and exact column norms, respectively.
<i>auxv</i>	Auxiliary vector.
<i>f</i>	Matrix $F' = L*Y'*A$.

?laqr0

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

```
call slaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
```

```
call dlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
```

```
call claqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
```

```
call zlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
```

Include Files

- `mkl.fi`

Description

The routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H=Z*T*Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q*H*Q^H = (QZ)*H*(QZ)^H$.

Input Parameters

<i>wantt</i>	LOGICAL. If <i>wantt</i> = <code>.TRUE.</code> , the full Schur form T is required; If <i>wantt</i> = <code>.FALSE.</code> , only eigenvalues are required.
<i>wantz</i>	LOGICAL. If <i>wantz</i> = <code>.TRUE.</code> , the matrix of Schur vectors Z is required; If <i>wantz</i> = <code>.FALSE.</code> , Schur vectors are not required.
<i>n</i>	INTEGER. The order of the Hessenberg matrix H . ($n \geq 0$).

ilo, ihi INTEGER.

It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$, and if $ilo > 1$ then $H(ilo, ilo-1) = 0$.

ilo and ihi are normally set by a previous call to `cgebal`, and then passed to `cgehrd` when the matrix output by `cgebal` is reduced to Hessenberg form. Otherwise, ilo and ihi should be set to 1 and n , respectively.

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$.

If $n=0$, then $ilo=1$ and $ihi=0$.

h REAL for `slaqr0`
 DOUBLE PRECISION for `dlaqr0`
 COMPLEX for `claqr0`
 DOUBLE COMPLEX for `zlaqr0`.

Array, DIMENSION (ldh, n), contains the upper Hessenberg matrix H .

ldh INTEGER. The leading dimension of the array h . $ldh \geq \max(1, n)$.

iloz, ihiz INTEGER. Specify the rows of Z to which transformations must be applied if *wantz* is `.TRUE.`, $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.

z REAL for `slaqr0`
 DOUBLE PRECISION for `dlaqr0`
 COMPLEX for `claqr0`
 DOUBLE COMPLEX for `zlaqr0`.

Array, DIMENSION (ldz, ihi), contains the matrix Z if *wantz* is `.TRUE.`. If *wantz* is `.FALSE.`, z is not referenced.

ldz INTEGER. The leading dimension of the array z .

If *wantz* is `.TRUE.`, then $ldz \geq \max(1, ihiz)$. Otherwise, $ldz \geq 1$.

work REAL for `slaqr0`
 DOUBLE PRECISION for `dlaqr0`
 COMPLEX for `claqr0`
 DOUBLE COMPLEX for `zlaqr0`.

Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.

$lwork \geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$.

It is recommended to use the workspace query to determine the optimal workspace size. If $lwork=-1$, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input

parameters n , ilo , and ihi . The estimate is returned in $work(1)$. No error messages related to the $lwork$ is issued by `xerbla`. Neither H nor Z are accessed.

Output Parameters

h	<p>If $info=0$, and $wantt$ is <code>.TRUE.</code>, then h contains the upper quasi-triangular/triangular matrix T from the Schur decomposition (the Schur form).</p> <p>If $info=0$, and $wantt$ is <code>.FALSE.</code>, then the contents of h are unspecified on exit.</p> <p>(The output values of h when $info > 0$ are given under the description of the $info$ parameter below.)</p> <p>The routine may explicitly set $h(i, j)$ for $i > j$ and $j=1, 2, \dots, ilo-1$ or $j=ihi+1, ihi+2, \dots, n$.</p>
$work(1)$	<p>On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.</p>
w	<p>COMPLEX for <code>claqr0</code></p> <p>DOUBLE COMPLEX for <code>zlaqr0</code>.</p> <p>Arrays, DIMENSION(n). The computed eigenvalues of $h(ilo:ihi, ilo:ihi)$ are stored in $w(ilo:ihi)$. If $wantt$ is <code>.TRUE.</code>, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h, with $w(i) = h(i, i)$.</p>
wr, wi	<p>REAL for <code>slaqr0</code></p> <p>DOUBLE PRECISION for <code>dlaqr0</code></p> <p>Arrays, DIMENSION(ihi) each. The real and imaginary parts, respectively, of the computed eigenvalues of $h(ilo:ihi, ilo:ihi)$ are stored in $wr(ilo:ihi)$ and $wi(ilo:ihi)$. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi, say the i-th and $(i+1)$-th, with $wi(i) > 0$ and $wi(i+1) < 0$. If $wantt$ is <code>.TRUE.</code>, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h, with $wr(i) = h(i, i)$, and if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, then $wi(i) = \sqrt{-h(i+1, i) * h(i, i+1)}$.</p>
z	<p>If $wantz$ is <code>.TRUE.</code>, then $z(ilo:ihi, iloz:ihiz)$ is replaced by $z(ilo:ihi, iloz:ihiz) * U$, where U is the orthogonal/unitary Schur factor of $h(ilo:ihi, ilo:ihi)$.</p> <p>If $wantz$ is <code>.FALSE.</code>, z is not referenced.</p> <p>(The output values of z when $info > 0$ are given under the description of the $info$ parameter below.)</p>
$info$	<p>INTEGER.</p> <p>= 0: the execution is successful.</p> <p>> 0: if $info = i$, then the routine failed to compute all the eigenvalues. Elements 1:$ilo-1$ and $i+1:n$ of wr and wi contain those eigenvalues which have been successfully computed.</p>

> 0: if *wantt* is `.FALSE.`, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is `.TRUE.`, then (initial value of *h*)**U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info+1* through *ihi*.

> 0: if *wantz* is `.TRUE.`, then (final value of $z(i_{lo}:i_{hi}, i_{loz}:i_{hiz})$)=(initial value of $z(i_{lo}:i_{hi}, i_{loz}:i_{hiz})$)**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is `.FALSE.`, then *z* is not accessed.

?laqr1

Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix *H* and specified shifts.

Syntax

```
call slaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call dlaqr1( n, h, ldh, sr1, si1, sr2, si2, v )
call claqr1( n, h, ldh, s1, s2, v )
call zlaqr1( n, h, ldh, s1, s2, v )
```

Include Files

- mkl.fi

Description

Given a 2-by-2 or 3-by-3 matrix *H*, this routine sets *v* to a scalar multiple of the first column of the product

$$K = (H - s1*I)*(H - s2*I), \text{ or } K = (H - (sr1 + i*si1)*I)*(H - (sr2 + i*si2)*I)$$

scaling to avoid overflows and most underflows.

It is assumed that either 1) $sr1 = sr2$ and $si1 = -si2$, or 2) $si1 = si2 = 0$.

This is useful for starting double implicit shift bulges in the QR algorithm.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>H</i> . <i>n</i> must be equal to 2 or 3.
<i>sr1, si2, sr2, si2</i>	REAL for slaqr1 DOUBLE PRECISION for dlaqr1 Shift values that define <i>K</i> in the formula above.
<i>s1, s2</i>	COMPLEX for claqr1 DOUBLE COMPLEX for zlaqr1.

Shift values that define K in the formula above.

h REAL for slaqr1
 DOUBLE PRECISION for dlaqr1
 COMPLEX for claqr1
 DOUBLE COMPLEX for zlaqr1.
 Array, DIMENSION (ldh, n), contains 2-by-2 or 3-by-3 matrix H in the formula above.

ldh INTEGER.
 The leading dimension of the array h just as declared in the calling routine.
 $ldh \geq n$.

Output Parameters

v REAL for slaqr1
 DOUBLE PRECISION for dlaqr1
 COMPLEX for claqr1
 DOUBLE COMPLEX for zlaqr1.
 Array with dimension (n).
 A scalar multiple of the first column of the matrix K in the formula above.

?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call dlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call claqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call zlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Include Files

- mkl.fi

Description

The routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

This subroutine is identical to `?laqr3` except that it avoids recursion by calling `?lahqr` instead of `?laqr4`.

Input Parameters

<code>wantt</code>	LOGICAL. If <code>wantt = .TRUE.</code> , then the Hessenberg matrix H is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine). If <code>wantt = .FALSE.</code> , then only enough of H is updated to preserve the eigenvalues.
<code>wantz</code>	LOGICAL. If <code>wantz = .TRUE.</code> , then the orthogonal/unitary matrix Z is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine). If <code>wantz = .FALSE.</code> , then Z is not referenced.
<code>n</code>	INTEGER. The order of the Hessenberg matrix H and (if <code>wantz = .TRUE.</code>) the order of the orthogonal/unitary matrix Z .
<code>ktop</code>	INTEGER. It is assumed that either <code>ktop=1</code> or <code>h(ktop,ktop-1)=0</code> . <code>ktop</code> and <code>kbot</code> together determine an isolated block along the diagonal of the Hessenberg matrix.
<code>kbot</code>	INTEGER. It is assumed without a check that either <code>kbot=n</code> or <code>h(kbot+1,kbot)=0</code> . <code>ktop</code> and <code>kbot</code> together determine an isolated block along the diagonal of the Hessenberg matrix.
<code>nw</code>	INTEGER. Size of the deflation window. $1 \leq nw \leq (kbot-ktop+1)$.
<code>h</code>	REAL for <code>slaqr2</code> DOUBLE PRECISION for <code>dlaqr2</code> COMPLEX for <code>claqr2</code> DOUBLE COMPLEX for <code>zlaqr2</code> . Array, DIMENSION (<code>ldh</code> , <code>n</code>), on input the initial n -by- n section of h stores the Hessenberg matrix H undergoing aggressive early deflation.
<code>ldh</code>	INTEGER. The leading dimension of the array h just as declared in the calling subroutine. $ldh \geq n$.
<code>iloz, ihiz</code>	INTEGER. Specify the rows of Z to which transformations must be applied if <code>wantz</code> is <code>.TRUE.</code> . $1 \leq iloz \leq ihiz \leq n$.

<i>z</i>	<p>REAL for slaqr2</p> <p>DOUBLE PRECISION for dlaqr2</p> <p>COMPLEX for claqr2</p> <p>DOUBLE COMPLEX for zlaqr2.</p> <p>Array, DIMENSION (<i>ldz</i>, <i>n</i>), contains the matrix <i>Z</i> if <i>wantz</i> is <code>.TRUE.</code>. If <i>wantz</i> is <code>.FALSE.</code>, then <i>z</i> is not referenced.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling subroutine. $ldz \geq 1$.</p>
<i>v</i>	<p>REAL for slaqr2</p> <p>DOUBLE PRECISION for dlaqr2</p> <p>COMPLEX for claqr2</p> <p>DOUBLE COMPLEX for zlaqr2.</p> <p>Workspace array with dimension (<i>ldv</i>, <i>nw</i>). An <i>nw</i>-by-<i>nw</i> work array.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling subroutine. $ldv \geq nw$.</p>
<i>nh</i>	<p>INTEGER. The number of column of <i>t</i>. $nh \geq nw$.</p>
<i>t</i>	<p>REAL for slaqr2</p> <p>DOUBLE PRECISION for dlaqr2</p> <p>COMPLEX for claqr2</p> <p>DOUBLE COMPLEX for zlaqr2.</p> <p>Workspace array with dimension (<i>ldt</i>, <i>nw</i>).</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of the array <i>t</i> just as declared in the calling subroutine. $ldt \geq nw$.</p>
<i>nv</i>	<p>INTEGER. The number of rows of work array <i>wv</i> available for workspace. $nv \geq nw$.</p>
<i>wv</i>	<p>REAL for slaqr2</p> <p>DOUBLE PRECISION for dlaqr2</p> <p>COMPLEX for claqr2</p> <p>DOUBLE COMPLEX for zlaqr2.</p> <p>Workspace array with dimension (<i>ldwv</i>, <i>nw</i>).</p>
<i>ldwv</i>	<p>INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling subroutine. $ldwv \geq nw$.</p>
<i>work</i>	<p>REAL for slaqr2</p> <p>DOUBLE PRECISION for dlaqr2</p> <p>COMPLEX for claqr2</p>

DOUBLE COMPLEX for `zlaqr2`.

Workspace array with dimension `lwork`.

`lwork`

INTEGER. The dimension of the array `work`.

`lwork=2*nw` is sufficient, but for the optimal performance a greater workspace may be required.

If `lwork=-1`, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters `n`, `nw`, `ktop`, and `kbot`. The estimate is returned in `work(1)`. No error messages related to the `lwork` is issued by `xerbla`. Neither `H` nor `Z` are accessed.

Output Parameters

`h`

On output `h` has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

`work(1)`

On exit `work(1)` is set to an estimate of the optimal value of `lwork` for the given values of the input parameters `n`, `nw`, `ktop`, and `kbot`.

`z`

If `wantz` is `.TRUE.`, then the orthogonal/unitary similarity transformation is accumulated into `z(ilo:ihiz, ilo:ihi)` from the right.

If `wantz` is `.FALSE.`, then `z` is unreferenced.

`nd`

INTEGER. The number of converged eigenvalues uncovered by the routine.

`ns`

INTEGER. The number of unconverged, that is approximate eigenvalues returned in `sr`, `si` or in `sh` that may be used as shifts by the calling subroutine.

`sh`

COMPLEX for `claqr2`

DOUBLE COMPLEX for `zlaqr2`.

Arrays, DIMENSION (`kbot`).

The approximate eigenvalues that may be used for shifts are stored in the `sh(kbot-nd-ns+1)` through the `sh(kbot-nd)`.

The converged eigenvalues are stored in the `sh(kbot-nd+1)` through the `sh(kbot)`.

`sr, si`

REAL for `slaqr2`

DOUBLE PRECISION for `dlaqr2`

Arrays, DIMENSION (`kbot`) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the `sr(kbot-nd-ns+1)` through the `sr(kbot-nd)`, and `si(kbot-nd-ns+1)` through the `si(kbot-nd)`, respectively.

The real and imaginary parts of converged eigenvalues are stored in the `sr(kbot-nd+1)` through the `sr(kbot)`, and `si(kbot-nd+1)` through the `si(kbot)`, respectively.

?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call dlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call claqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

```
call zlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Include Files

- mkl.fi

Description

The routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

Input Parameters

<code>wantt</code>	LOGICAL. If <code>wantt = .TRUE.</code> , then the Hessenberg matrix H is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine). If <code>wantt = .FALSE.</code> , then only enough of H is updated to preserve the eigenvalues.
<code>wantz</code>	LOGICAL. If <code>wantz = .TRUE.</code> , then the orthogonal/unitary matrix Z is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine). If <code>wantz = .FALSE.</code> , then Z is not referenced.
<code>n</code>	INTEGER. The order of the Hessenberg matrix H and (if <code>wantz = .TRUE.</code>) the order of the orthogonal/unitary matrix Z .
<code>ktop</code>	INTEGER. It is assumed that either <code>ktop=1</code> or <code>h(ktop,ktop-1)=0</code> . <code>ktop</code> and <code>kbot</code> together determine an isolated block along the diagonal of the Hessenberg matrix.

<i>kbot</i>	<p>INTEGER.</p> <p>It is assumed without a check that either $kbot=n$ or $h(kbot+1, kbot)=0$. <i>ktop</i> and <i>kbot</i> together determine an isolated block along the diagonal of the Hessenberg matrix.</p>
<i>nw</i>	<p>INTEGER.</p> <p>Size of the deflation window. $1 \leq nw \leq (kbot - ktop + 1)$.</p>
<i>h</i>	<p>REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 DOUBLE COMPLEX for zlaqr3.</p> <p>Array, DIMENSION (<i>ldh</i>, <i>n</i>), on input the initial <i>n</i>-by-<i>n</i> section of <i>h</i> stores the Hessenberg matrix <i>H</i> undergoing aggressive early deflation.</p>
<i>ldh</i>	<p>INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling subroutine. $ldh \geq n$.</p>
<i>iloz, ihiz</i>	<p>INTEGER. Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code>. $1 \leq iloz \leq ihiz \leq n$.</p>
<i>z</i>	<p>REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 DOUBLE COMPLEX for zlaqr3.</p> <p>Array, DIMENSION (<i>ldz</i>, <i>n</i>), contains the matrix <i>Z</i> if <i>wantz</i> is <code>.TRUE.</code>. If <i>wantz</i> is <code>.FALSE.</code>, then <i>z</i> is not referenced.</p>
<i>ldz</i>	<p>INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling subroutine. $ldz \geq 1$.</p>
<i>v</i>	<p>REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 DOUBLE COMPLEX for zlaqr3.</p> <p>Workspace array with dimension (<i>ldv</i>, <i>nw</i>). An <i>nw</i>-by-<i>nw</i> work array.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling subroutine. $ldv \geq nw$.</p>
<i>nh</i>	<p>INTEGER. The number of column of <i>t</i>. $nh \geq nw$.</p>
<i>t</i>	<p>REAL for slaqr3 DOUBLE PRECISION for dlaqr3 COMPLEX for claqr3 DOUBLE COMPLEX for zlaqr3.</p>

Workspace array with dimension (ldt, nw) .

ldt INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. $ldt \geq nw$.

nv INTEGER. The number of rows of work array *wv* available for workspace. $nv \geq nw$.

wv REAL for slaqr3
 DOUBLE PRECISION for dlaqr3
 COMPLEX for claqr3
 DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension $(ldwv, nw)$.

ldwv INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. $ldwv \geq nw$.

work REAL for slaqr3
 DOUBLE PRECISION for dlaqr3
 COMPLEX for claqr3
 DOUBLE COMPLEX for zlaqr3.

Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.
lwork=2*nw) is sufficient, but for the optimal performance a greater workspace may be required.

If *lwork*=-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by xerbla. Neither *H* nor *Z* are accessed.

Output Parameters

h On output *h* has been transformed by an orthogonal/unitary similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

work(1) On exit *work*(1) is set to an estimate of the optimal value of *lwork* for the given values of the input parameters *n*, *nw*, *ktop*, and *kbot*.

z If *wantz* is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into *z*(*iloz:ihiz*, *ilo:ihi*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

nd INTEGER. The number of converged eigenvalues uncovered by the routine.

ns INTEGER. The number of unconverged, that is approximate eigenvalues returned in *sr*, *si* or in *sh* that may be used as shifts by the calling subroutine.

<i>sh</i>	<p>COMPLEX for <code>claqr3</code></p> <p>DOUBLE COMPLEX for <code>zlaqr3</code>.</p> <p>Arrays, DIMENSION (<i>kbot</i>).</p> <p>The approximate eigenvalues that may be used for shifts are stored in the <code>sh(kbot-nd-ns+1)</code> through the <code>sh(kbot-nd)</code>.</p> <p>The converged eigenvalues are stored in the <code>sh(kbot-nd+1)</code> through the <code>sh(kbot)</code>.</p>
<i>sr, si</i>	<p>REAL for <code>slaqr3</code></p> <p>DOUBLE PRECISION for <code>dlaqr3</code></p> <p>Arrays, DIMENSION (<i>kbot</i>) each.</p> <p>The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the <code>sr(kbot-nd-ns+1)</code> through the <code>sr(kbot-nd)</code>, and <code>si(kbot-nd-ns+1)</code> through the <code>si(kbot-nd)</code>, respectively.</p> <p>The real and imaginary parts of converged eigenvalues are stored in the <code>sr(kbot-nd+1)</code> through the <code>sr(kbot)</code>, and <code>si(kbot-nd+1)</code> through the <code>si(kbot)</code>, respectively.</p>

?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

```
call slaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
```

```
call dlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
```

```
call claqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
```

```
call zlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
```

Include Files

- `mkl.fi`

Description

The routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H = Z^* T^* Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q^* H^* Q^H = (QZ)^* H^* (QZ)^H$.

This routine implements one level of recursion for `?laqr0`. It is a complete implementation of the small bulge multi-shift QR algorithm. It may be called by `?laqr0` and, for large enough deflation window size, it may be called by `?laqr3`. This routine is identical to `?laqr0` except that it calls `?laqr2` instead of `?laqr3`.

Input Parameters

<i>wantt</i>	LOGICAL. If <i>wantt</i> = <code>.TRUE.</code> , the full Schur form <i>T</i> is required; If <i>wantt</i> = <code>.FALSE.</code> , only eigenvalues are required.
<i>wantz</i>	LOGICAL. If <i>wantz</i> = <code>.TRUE.</code> , the matrix of Schur vectors <i>Z</i> is required; If <i>wantz</i> = <code>.FALSE.</code> , Schur vectors are not required.
<i>n</i>	INTEGER. The order of the Hessenberg matrix <i>H</i> . ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. It is assumed that <i>H</i> is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$, and if $ilo > 1$ then $h(ilo, ilo-1) = 0$. <i>ilo</i> and <i>ihi</i> are normally set by a previous call to <code>cgebal</code> , and then passed to <code>cgehrd</code> when the matrix output by <code>cgebal</code> is reduced to Hessenberg form. Otherwise, <i>ilo</i> and <i>ihi</i> should be set to 1 and <i>n</i> , respectively. If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n=0$, then $ilo=1$ and $ihi=0$
<i>h</i>	REAL for <code>slaqr4</code> DOUBLE PRECISION for <code>dlaqr4</code> COMPLEX for <code>claqr4</code> DOUBLE COMPLEX for <code>zlaqr4</code> . Array, DIMENSION (<i>ldh</i> , <i>n</i>), contains the upper Hessenberg matrix <i>H</i> .
<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> . $ldh \geq \max(1, n)$.
<i>iloz, ihiz</i>	INTEGER. Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code> , $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.
<i>z</i>	REAL for <code>slaqr4</code> DOUBLE PRECISION for <code>dlaqr4</code> COMPLEX for <code>claqr4</code> DOUBLE COMPLEX for <code>zlaqr4</code> . Array, DIMENSION (<i>ldz</i> , <i>ihiz</i>), contains the matrix <i>Z</i> if <i>wantz</i> is <code>.TRUE.</code> . If <i>wantz</i> is <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> . If <i>wantz</i> is <code>.TRUE.</code> , then $ldz \geq \max(1, ihiz)$. Otherwise, $ldz \geq 1$.
<i>work</i>	REAL for <code>slaqr4</code> DOUBLE PRECISION for <code>dlaqr4</code> COMPLEX for <code>claqr4</code>

DOUBLE COMPLEX for `zlaqr4`.

Workspace array with dimension *lwork*.

lwork

INTEGER. The dimension of the array *work*.

$lwork \geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6 * n$.

It is recommended to use the workspace query to determine the optimal workspace size. If *lwork* = -1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n*, *ilo*, and *ihi*. The estimate is returned in *work*(1). No error messages related to the *lwork* is issued by `xerbla`. Neither *H* nor *Z* are accessed.

Output Parameters

h

If *info* = 0, and *wantt* is `.TRUE.`, then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form).

If *info* = 0, and *wantt* is `.FALSE.`, then the contents of *h* are unspecified on exit.

(The output values of *h* when *info* > 0 are given under the description of the *info* parameter below.)

The routines may explicitly set *h*(*i*, *j*) for *i* > *j* and *j* = 1, 2, ... *ilo* - 1 or *j* = *ihi* + 1, *ihi* + 2, ... *n*.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

w

COMPLEX for `claqr4`

DOUBLE COMPLEX for `zlaqr4`.

Arrays, DIMENSION(*n*). The computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in *w*(*ilo*:*ihi*). If *wantt* is `.TRUE.`, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *w*(*i*) = *h*(*i*, *i*).

wr, *wi*

REAL for `slaqr4`

DOUBLE PRECISION for `dlaqr4`

Arrays, DIMENSION(*ihi*) each. The real and imaginary parts, respectively, of the computed eigenvalues of *h*(*ilo*:*ihi*, *ilo*:*ihi*) are stored in the *wr*(*ilo*:*ihi*) and *wi*(*ilo*:*ihi*). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi*(*i*) > 0 and *wi*(*i*+1) < 0. If *wantt* is `.TRUE.`, then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr*(*i*) = *h*(*i*, *i*), and if *h*(*i*:*i*+1, *i*:*i*+1) is a 2-by-2 diagonal block, then *wi*(*i*) = $\sqrt{-h(i+1, i) * h(i, i+1)}$.

z

If *wantz* is `.TRUE.`, then *z*(*ilo*:*ihi*, *iloz*:*ihiz*) is replaced by *z*(*ilo*:*ihi*, *iloz*:*ihiz*)**U*, where *U* is the orthogonal/unitary Schur factor of *h*(*ilo*:*ihi*, *ilo*:*ihi*).

If *wantz* is `.FALSE.`, *z* is not referenced.

(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info

INTEGER.

= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues. Elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt* is `.FALSE.`, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is `.TRUE.`, then (initial value of *h*)**U* = *U**(final value of *h*, where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is `.TRUE.`, then (final value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))=(initial value of *z*(*ilo*:*ihi*, *iloz*:*ihiz*))**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is `.FALSE.`, then *z* is not accessed.

?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call slaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz,
z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

```
call dlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz,
z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

```
call claqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z,
ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

```
call zlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z,
ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Include Files

- mkl.fi

Description

This auxiliary routine called by ?laqr0 performs a single small-bulge multi-shift QR sweep.

Input Parameters

wantt LOGICAL.

	<i>wantt</i> = .TRUE. if the quasi-triangular/triangular Schur factor is computed. <i>wantt</i> is set to .FALSE. otherwise.
<i>wantz</i>	LOGICAL. <i>wantz</i> = .TRUE. if the orthogonal/unitary Schur factor is computed. <i>wantz</i> is set to .FALSE. otherwise.
<i>kacc22</i>	INTEGER. Possible values are 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: the routine does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: the routine accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: the routine accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<i>n</i>	INTEGER. The order of the Hessenberg matrix <i>H</i> upon which the routine operates.
<i>ktop, kbot</i>	INTEGER. It is assumed without a check that either <i>ktop</i> =1 or $h(ktop, ktop-1)=0$, and either <i>kbot</i> = <i>n</i> or $h(kbot+1, kbot)=0$.
<i>nshfts</i>	INTEGER. Number of simultaneous shifts, must be positive and even.
<i>sr, si</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 Arrays, DIMENSION (<i>nshfts</i>) each. <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.
<i>s</i>	COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Arrays, DIMENSION (<i>nshfts</i>). <i>s</i> contains the shifts of origin that define the multi-shift QR sweep.
<i>h</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Array, DIMENSION (<i>ldh, n</i>), on input contains the Hessenberg matrix.

<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling routine. $ldh \geq \max(1, n)$.
<i>iloz, ihiz</i>	INTEGER. Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code> . $1 \leq iloz \leq ihiz \leq n$.
<i>z</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Array, DIMENSION (<i>ldz</i> , <i>ihi</i>), contains the matrix <i>Z</i> if <i>wantz</i> is <code>.TRUE.</code> . If <i>wantz</i> is <code>.FALSE.</code> , then <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling routine. $ldz \geq n$.
<i>v</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension (<i>ldv</i> , $nshfts/2$).
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling routine. $ldv \geq 3$.
<i>u</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension (<i>ldu</i> , $3*nshfts-3$).
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> just as declared in the calling routine. $ldu \geq 3*nshfts-3$.
<i>nh</i>	INTEGER. The number of column in the array <i>wh</i> available for workspace. $nh \geq 1$.
<i>wh</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension (<i>ldwh</i> , <i>nh</i>)
<i>ldwh</i>	INTEGER. The leading dimension of the array <i>wh</i> just as declared in the calling routine. $ldwh \geq 3*nshfts-3$

<i>nv</i>	INTEGER. The number of rows of the array <i>wv</i> available for workspace. $nv \geq 1$.
<i>wv</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension $(ldwv, 3*nshfts-3)$.
<i>ldwv</i>	INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling routine. $ldwv \geq nv$.

Output Parameters

<i>sr, si</i>	On output, may be reordered.
<i>h</i>	On output a multi-shift QR Sweep with shifts $sr(j)+i*si(j)$ or $s(j)$ is applied to the isolated diagonal block in rows and columns <i>ktop</i> through <i>kbot</i> .
<i>z</i>	If <i>wantz</i> is <code>.TRUE.</code> , then the QR Sweep orthogonal/unitary similarity transformation is accumulated into $z(i_{loz}:i_{hiz}, i_{lo}:i_{hi})$ from the right. If <i>wantz</i> is <code>.FALSE.</code> , then <i>z</i> is unreferenced.

?laqsb

Scales a symmetric band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- `mkl.fi`

Description

The routine equilibrates a symmetric band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular.
-------------	---

	If <code>uplo = 'L'</code> : lower triangular.
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$.
<code>kd</code>	INTEGER. The number of super-diagonals of the matrix A if <code>uplo = 'U'</code> , or the number of sub-diagonals if <code>uplo = 'L'</code> . $kd \geq 0$.
<code>ab</code>	REAL for <code>slaqsb</code> DOUBLE PRECISION for <code>dlaqsb</code> COMPLEX for <code>claqsb</code> DOUBLE COMPLEX for <code>zlaqsb</code> Array, DIMENSION ($ldab, n$). On entry, the upper or lower triangle of the symmetric band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array <code>ab</code> as follows: if <code>uplo = 'U'</code> , $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$; if <code>uplo = 'L'</code> , $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.
<code>ldab</code>	INTEGER. The leading dimension of the array <code>ab</code> . $ldab \geq kd+1$.
<code>s</code>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Array, DIMENSION (n). The scale factors for A .
<code>scond</code>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Ratio of the smallest $s(i)$ to the largest $s(i)$.
<code>amax</code>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Absolute value of largest matrix entry.

Output Parameters

<code>ab</code>	On exit, if <code>info = 0</code> , the triangular factor U or L from the Cholesky factorization of the band matrix A that can be $A = U^T * U$ or $A = L * L^T$ for real flavors and $A = U^H * U$ or $A = L * L^H$ for complex flavors, in the same storage format as A .
<code>equed</code>	CHARACTER*1. Specifies whether or not equilibration was done. If <code>equed = 'N'</code> : No equilibration.

If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $s_{\text{cond}} < \text{thresh}$, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.

Syntax

```
call slaqsp( uplo, n, ap, s, scond, amax, equed )
call dlaqsp( uplo, n, ap, s, scond, amax, equed )
call claqsp( uplo, n, ap, s, scond, amax, equed )
call zlaqsp( uplo, n, ap, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine ?laqsp equilibrates a symmetric matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>ap</i>	REAL for slaqsp DOUBLE PRECISION for dlaqsp COMPLEX for claqsp DOUBLE COMPLEX for zlaqsp Array, DIMENSION $(n(n+1)/2)$. On entry, the upper or lower triangle of the symmetric matrix <i>A</i> , packed columnwise in a linear array. The <i>j</i> -th column of <i>A</i> is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$;

if $uplo = 'L'$, $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

s REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Array, DIMENSION (*n*). The scale factors for A.

scond REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Ratio of the smallest $s(i)$ to the largest $s(i)$.

amax REAL for slaqsp/claqsp
 DOUBLE PRECISION for dlaqsp/zlaqsp
 Absolute value of largest matrix entry.

Output Parameters

ap On exit, the equilibrated matrix: $diag(s) * A * diag(s)$, in the same storage format as A.

equed CHARACTER*1.
 Specifies whether or not equilibration was done.
 If *equed* = 'N': No equilibration.
 If *equed* = 'Y': Equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.

Syntax

```
call slaqsy( uplo, n, a, lda, s, scnd, amax, equed )
call dlaqsy( uplo, n, a, lda, s, scnd, amax, equed )
call claqsy( uplo, n, a, lda, s, scnd, amax, equed )
call zlaqsy( uplo, n, a, lda, s, scnd, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a symmetric matrix A using the scaling factors in the vector s.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	REAL for <i>slaqsy</i> DOUBLE PRECISION for <i>dlaqsy</i> COMPLEX for <i>claqsy</i> DOUBLE COMPLEX for <i>zlaqsy</i> Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the symmetric matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(n, 1)$.
<i>s</i>	REAL for <i>slaqsy/claqsy</i> DOUBLE PRECISION for <i>dlaqsy/zlaqsy</i> Array, DIMENSION (<i>n</i>). The scale factors for <i>A</i> .
<i>scond</i>	REAL for <i>slaqsy/claqsy</i> DOUBLE PRECISION for <i>dlaqsy/zlaqsy</i> Ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for <i>slaqsy/claqsy</i> DOUBLE PRECISION for <i>dlaqsy/zlaqsy</i> Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	On exit, if <i>equed</i> = 'Y', the equilibrated matrix: $\text{diag}(s) * A * \text{diag}(s)$.
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If <i>equed</i> = 'N': No equilibration.

This routine is designed for the condition number estimation in routine `?trsna`.

Input Parameters

<i>ltran</i>	LOGICAL. On entry, <i>ltran</i> specifies the option of conjugate transpose: = <code>.FALSE.</code> , $\text{op}(T + iB) = T + iB$, = <code>.TRUE.</code> , $\text{op}(T + iB) = (T + iB)^T$.
<i>lreal</i>	LOGICAL. On entry, <i>lreal</i> specifies the input matrix structure: = <code>.FALSE.</code> , the input is complex = <code>.TRUE.</code> , the input is real.
<i>n</i>	INTEGER. On entry, <i>n</i> specifies the order of $T + iB$. $n \geq 0$.
<i>t</i>	REAL for <code>slaqtr</code> DOUBLE PRECISION for <code>dlaqtr</code> Array, dimension (ldt, n) . On entry, <i>t</i> contains a matrix in Schur canonical form. If <i>lreal</i> = <code>.FALSE.</code> , then the first diagonal block of <i>t</i> must be 1-by-1.
<i>ldt</i>	INTEGER. The leading dimension of the matrix <i>T</i> . $ldt \geq \max(1, n)$.
<i>b</i>	REAL for <code>slaqtr</code> DOUBLE PRECISION for <code>dlaqtr</code> Array, dimension (n) . On entry, <i>b</i> contains the elements to form the matrix <i>B</i> as described above. If <i>lreal</i> = <code>.TRUE.</code> , <i>b</i> is not referenced.
<i>w</i>	REAL for <code>slaqtr</code> DOUBLE PRECISION for <code>dlaqtr</code> On entry, <i>w</i> is the diagonal element of the matrix <i>B</i> . If <i>lreal</i> = <code>.TRUE.</code> , <i>w</i> is not referenced.
<i>x</i>	REAL for <code>slaqtr</code> DOUBLE PRECISION for <code>dlaqtr</code> Array, dimension $(2n)$. On entry, <i>x</i> contains the right hand side of the system.
<i>work</i>	REAL for <code>slaqtr</code> DOUBLE PRECISION for <code>dlaqtr</code> Workspace array, dimension (n) .

Output Parameters

<i>scale</i>	REAL for slaqtr DOUBLE PRECISION for dlaqtr On exit, <i>scale</i> is the scale factor.
<i>x</i>	On exit, <i>X</i> is overwritten by the solution.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = 1: the some diagonal 1-by-1 block has been perturbed by a small number <i>smin</i> to keep nonsingularity. If <i>info</i> = 2: the some diagonal 2-by-2 block has been perturbed by a small number in <code>?1aln2</code> to keep nonsingularity.

NOTE

For higher speed, this routine does not check the inputs for errors.

?lar1v

Computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of tridiagonal matrix.

Syntax

```
call slar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

```
call dlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

```
call clar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

```
call zlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

Include Files

- mkl.fi

Description

The routine ?lar1v computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $L^*D^*L^T - \lambda^*I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, *r* corresponds to the index where the eigenvector is largest in magnitude.

The following steps accomplish this computation :

- Stationary *qd* transform, $L^*D^*L^T - \lambda^*I = L(+)*D(+)*L(+)^T$
- Progressive *qd* transform, $L^*D^*L^T - \lambda^*I = U(-)*D(-)*U(-)^T$,

- Computation of the diagonal elements of the inverse of $L^*D^*L^T - \lambda^*I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.
- Computation of the (scaled) r -th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix $L^*D^*L^T$.
<i>b1</i>	INTEGER. First index of the submatrix of $L^*D^*L^T$.
<i>bn</i>	INTEGER. Last index of the submatrix of $L^*D^*L^T$.
<i>lambda</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v The shift. To compute an accurate eigenvector, <i>lambda</i> should be a good approximation to an eigenvalue of $L^*D^*L^T$.
<i>l</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION ($n-1$). The ($n-1$) subdiagonal elements of the unit bidiagonal matrix L , in elements 1 to $n-1$.
<i>d</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D .
<i>ld</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION ($n-1$). The $n-1$ elements $L_i^*D_i$.
<i>lld</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v Array, DIMENSION ($n-1$). The $n-1$ elements $L_i^*L_i^*D_i$.
<i>pivmin</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v The minimum pivot in the Sturm sequence.
<i>gaptol</i>	REAL for slar1v/cclar1v DOUBLE PRECISION for dlar1v/zlar1v

Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z REAL for `slar1v`
 DOUBLE PRECISION for `dlar1v`
 COMPLEX for `clar1v`
 DOUBLE COMPLEX for `zlar1v`
 Array, DIMENSION (*n*). All entries of *z* must be set to 0.

wantnc LOGICAL.
 Specifies whether *negcnt* has to be computed.

r INTEGER.
 The twist index for the twisted factorization used to compute *z*. On input, $0 \leq r \leq n$. If *r* is input as 0, *r* is set to the index where $(L * D * L^T - \lambda * I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, *r* is unchanged.

work REAL for `slar1v/clar1v`
 DOUBLE PRECISION for `dlar1v/zlar1v`
 Workspace array, DIMENSION ($4 * n$).

Output Parameters

z REAL for `slar1v`
 DOUBLE PRECISION for `dlar1v`
 COMPLEX for `clar1v`
 DOUBLE COMPLEX for `zlar1v`
 Array, DIMENSION (*n*). The (scaled) *r*-th column of the inverse. *z*(*r*) is returned to be 1.

negcnt INTEGER. If *wantnc* is `.TRUE.` then *negcnt* = the number of pivots < *pivmin* in the matrix factorization $L * D * L^T$, and *negcnt* = -1 otherwise.

ztz REAL for `slar1v/clar1v`
 DOUBLE PRECISION for `dlar1v/zlar1v`
 The square of the 2-norm of *z*.

mingma REAL for `slar1v/clar1v`
 DOUBLE PRECISION for `dlar1v/zlar1v`
 The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L * D * L^T - \lambda * I$.

r On output, *r* is the twist index used to compute *z*. Ideally, *r* designates the position of the maximum entry in the eigenvector.

isuppz INTEGER. Array, DIMENSION (2). The support of the vector in *Z*, that is, the vector *z* is nonzero only in elements *isuppz*(1) through *isuppz*(2).

nrminv REAL for slar1v/clar1v
 DOUBLE PRECISION for dlar1v/zlar1v
 Equals $1/\sqrt{ztz}$.

resid REAL for slar1v/clar1v
 DOUBLE PRECISION for dlar1v/zlar1v
 The residual of the FP vector.
 $resid = \text{ABS}(mingma)/\sqrt{ztz}$.

rqcorr REAL for slar1v/clar1v
 DOUBLE PRECISION for dlar1v/zlar1v
 The Rayleigh Quotient correction to *lambda*.
 $rqcorr = mingma/ztz$.

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v( n, x, y, z, incx, c, s, incc )
call dlar2v( n, x, y, z, incx, c, s, incc )
call clar2v( n, x, y, z, incx, c, s, incc )
call zlar2v( n, x, y, z, incx, c, s, incc )
```

Include Files

- mkl.fi

Description

The routine ?lar2v applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors *x*, *y* and *z*. For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \text{conjg}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \text{conjg}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\text{conjg}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

n INTEGER. The number of plane rotations to be applied.

x, y, z REAL for slar2v
 DOUBLE PRECISION for dlar2v
 COMPLEX for clar2v

DOUBLE COMPLEX for `zlar2v`

Arrays, DIMENSION $(1+(n-1)*incx)$ each. Contain the vectors `x`, `y` and `z`, respectively. For all flavors of `?lar2v`, elements of `x` and `y` are assumed to be real.

`incx` INTEGER. The increment between elements of `x`, `y`, and `z`. $incx > 0$.

`c` REAL for `slar2v/clar2v`

DOUBLE PRECISION for `dlar2v/zlar2v`

Array, DIMENSION $(1+(n-1)*incc)$. The cosines of the plane rotations.

`s` REAL for `slar2v`

DOUBLE PRECISION for `dlar2v`

COMPLEX for `clar2v`

DOUBLE COMPLEX for `zlar2v`

Array, DIMENSION $(1+(n-1)*incc)$. The sines of the plane rotations.

`incc` INTEGER. The increment between elements of `c` and `s`. $incc > 0$.

Output Parameters

`x`, `y`, `z` Vectors `x`, `y` and `z`, containing the results of transform.

?laran

Returns a random real number from a uniform distribution.

Syntax

```
res = slaran (iseed)
```

```
res = dlaran (iseed)
```

Description

The `?laran` routine returns a random real number from a uniform (0,1) distribution. This routine uses a multiplicative congruential method with modulus 2^{48} and multiplier 33952834046453. 48-bit integers are stored in four integer array elements with 12 bits per element. Hence the routine is portable across machines with integers of 32 bits or more.

Input Parameters

`iseed` INTEGER. Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and `iseed(4)` must be odd.

Output Parameters

`iseed` INTEGER.
On exit, the seed is updated.

res REAL for slarf,
DOUBLE PRECISION for dlarf,
Random number.

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

call slarf(*side*, *m*, *n*, *v*, *incv*, *tau*, *c*, *ldc*, *work*)

call dlarf(*side*, *m*, *n*, *v*, *incv*, *tau*, *c*, *ldc*, *work*)

call clarf(*side*, *m*, *n*, *v*, *incv*, *tau*, *c*, *ldc*, *work*)

call zlarf(*side*, *m*, *n*, *v*, *incv*, *tau*, *c*, *ldc*, *work*)

Include Files

- mkl.fi

Description

The routine applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in one of the following forms:

- $H = I - \tau v v^T$

where τ is a real scalar and v is a real vector.

If $\tau = 0$, then H is taken to be the unit matrix.

- $H = I - \tau v v^H$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then H is taken to be the unit matrix. For clarf/zlarf, to apply H^H (the conjugate transpose of H), supply $\text{conjg}(\tau)$ instead of τ .

Input Parameters

side CHARACTER*1.
If *side* = 'L': form $H*C$
If *side* = 'R': form $C*H$.

m INTEGER. The number of rows of the matrix C .

n INTEGER. The number of columns of the matrix C .

v REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
DOUBLE COMPLEX for zlarf
Array, DIMENSION

$(1 + (m-1) * \text{abs}(incv))$ if *side* = 'L' or
 $(1 + (n-1) * \text{abs}(incv))$ if *side* = 'R'. The vector *v* in the representation of *H*. *v* is not used if *tau* = 0.

incv INTEGER. The increment between elements of *v*.
incv ≠ 0.

tau REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
DOUBLE COMPLEX for zlarf
The value *tau* in the representation of *H*.

c REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
DOUBLE COMPLEX for zlarf
Array, DIMENSION (*ldc*,*n*).
On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.
ldc ≥ max(1, *m*).

work REAL for slarf
DOUBLE PRECISION for dlarf
COMPLEX for clarf
DOUBLE COMPLEX for zlarf
Workspace array, DIMENSION
(*n*) if *side* = 'L' or
(*m*) if *side* = 'R'.

Output Parameters

c On exit, *C* is overwritten by the matrix H^*C if *side* = 'L', or C^*H if *side* = 'R'.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

call slarfb(*side*, *trans*, *direct*, *storev*, *m*, *n*, *k*, *v*, *ldv*, *t*, *ldt*, *c*, *ldc*, *work*, *ldwork*)

```
call dlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

```
call clarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

```
call zlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

Include Files

- mkl.fi

Description

The real flavors of the routine `?larfb` apply a real block reflector H or its transpose H^T to a real m -by- n matrix C from either left or right.

The complex flavors of the routine `?larfb` apply a complex block reflector H or its conjugate transpose H^H to a complex m -by- n matrix C from either left or right.

Input Parameters

The data types are given for the Fortran interface.

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': apply H or H^T for real flavors and H or H^H for complex flavors from the left. If <i>side</i> = 'R': apply H or H^T for real flavors and H or H^H for complex flavors from the right.
<i>trans</i>	CHARACTER*1. If <i>trans</i> = 'N': apply H (No transpose). If <i>trans</i> = 'C': apply H^H (Conjugate transpose). If <i>trans</i> = 'T': apply H^T (Transpose).
<i>direct</i>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors If <i>direct</i> = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward) If <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Indicates how the vectors which define the elementary reflectors are stored: If <i>storev</i> = 'C': Column-wise If <i>storev</i> = 'R': Row-wise
<i>m</i>	INTEGER. The number of rows of the matrix C .
<i>n</i>	INTEGER. The number of columns of the matrix C .
<i>k</i>	INTEGER. The order of the matrix T (equal to the number of elementary reflectors whose product defines the block reflector).

<i>v</i>	<p>REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb DOUBLE COMPLEX for zlarfb</p> <p>Array, DIMENSION <i>(ldv, k)</i> if <i>storev</i> = 'C' <i>(ldv, m)</i> if <i>storev</i> = 'R' and <i>side</i> = 'L' <i>(ldv, n)</i> if <i>storev</i> = 'R' and <i>side</i> = 'R'</p> <p>The matrix <i>v</i>. See <i>Application Notes</i> below.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i>. If <i>storev</i> = 'C' and <i>side</i> = 'L', $ldv \geq \max(1, m)$; if <i>storev</i> = 'C' and <i>side</i> = 'R', $ldv \geq \max(1, n)$; if <i>storev</i> = 'R', $ldv \geq k$.</p>
<i>t</i>	<p>REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb DOUBLE COMPLEX for zlarfb</p> <p>Array, size <i>(ldt, k)</i>.</p> <p>Contains the triangular <i>k</i>-by-<i>k</i> matrix <i>T</i> in the representation of the block reflector.</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of the array <i>t</i>. $ldt \geq k$.</p>
<i>c</i>	<p>REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb DOUBLE COMPLEX for zlarfb</p> <p>Array, size <i>(ldc, n)</i>.</p> <p>On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.</p>
<i>ldc</i>	<p>INTEGER. The leading dimension of the array <i>c</i>. $ldc \geq \max(1, m)$.</p>
<i>work</i>	<p>REAL for slarfb DOUBLE PRECISION for dlarfb COMPLEX for clarfb DOUBLE COMPLEX for zlarfb</p> <p>Workspace array, DIMENSION <i>(ldwork, k)</i>.</p>

ldwork INTEGER. The leading dimension of the array *work*.
 If *side* = 'L', $ldwork \geq \max(1, n)$;
 if *side* = 'R', $ldwork \geq \max(1, m)$.

Output Parameters

c On exit, *c* is overwritten by the product of the following:

- H^*C , or H^T*C , or C^*H , or C^*H^T for real flavors
- H^*C , or H^H*C , or C^*H , or C^*H^H for complex flavors

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = -1011, memory allocation error occurred.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.



direct = 'B' and *storev* = 'C':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg( n, alpha, x, incx, tau )
call dlarfg( n, alpha, x, incx, tau )
call clarfg( n, alpha, x, incx, tau )
call zlarfg( n, alpha, x, incx, tau )
```

Include Files

- mkl.fi

Description

The routine ?larfg generates a real/complex elementary reflector H of order n , such that

$$H^* \begin{bmatrix} \alpha \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^T * H = I, \quad \text{for real flavors and}$$

$$H^X * \begin{bmatrix} \alpha \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^X * H = I, \quad \text{for complex flavors,}$$

where α and β are scalars (with β real for all flavors), and \mathbf{x} is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau \mathbf{x}^* \begin{bmatrix} 1 \\ \mathbf{v} \end{bmatrix} * \begin{bmatrix} 1 & \mathbf{v}^T \end{bmatrix} \quad \text{for real flavors and}$$

$$H = I - \tau \mathbf{x}^* \begin{bmatrix} 1 \\ \mathbf{v} \end{bmatrix} * \begin{bmatrix} 1 & \mathbf{v}^H \end{bmatrix} \quad \text{for complex flavors,}$$

where τ is a real/complex scalar and \mathbf{v} is a real/complex $(n-1)$ -element vector, respectively. Note that for clarfg/zlarfg, H is not Hermitian.

If the elements of \mathbf{x} are all zero (and, for complex flavors, α is real), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau \leq 2$ (for real flavors), or

$1 \leq \text{Re}(\tau) \leq 2$ and $|\text{abs}(\tau - 1)| \leq 1$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

n	INTEGER. The order of the elementary reflector.
α	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg

	DOUBLE COMPLEX for zlarfg On entry, the value <i>alpha</i> .
<i>x</i>	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg DOUBLE COMPLEX for zlarfg Array, size $(1+(n-2)*abs(incx))$. On entry, the vector <i>x</i> .
<i>incx</i>	INTEGER. The increment between elements of <i>x</i> . $incx > 0$.

Output Parameters

<i>alpha</i>	On exit, it is overwritten with the value <i>beta</i> .
<i>x</i>	On exit, it is overwritten with the vector <i>v</i> .
<i>tau</i>	REAL for slarfg DOUBLE PRECISION for dlarfg COMPLEX for clarfg DOUBLE COMPLEX for zlarfg The value <i>tau</i> .

?larfgp

Generates an elementary reflector (Householder matrix) with non-negative beta .

Syntax

```
call slarfgp( n, alpha, x, incx, tau )
call dlarfgp( n, alpha, x, incx, tau )
call clarfgp( n, alpha, x, incx, tau )
call zlarfgp( n, alpha, x, incx, tau )
```

Include Files

- mkl.fi

Description

The routine ?larfgp generates a real/complex elementary reflector *H* of order *n*, such that

$$H^* \begin{bmatrix} \alpha \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^T * H = I, \quad \text{for real flavors and}$$

$$H^{X*} \begin{bmatrix} \alpha \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^{X*} * H = I, \quad \text{for complex flavors,}$$

where α and β are scalars (with β real and non-negative for all flavors), and x is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau \alpha x^* \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix} \quad \text{for real flavors and}$$

$$H = I - \tau \alpha x^* \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^H \end{bmatrix} \quad \text{for complex flavors,}$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that for `c/zlarfgp`, H is not Hermitian.

If the elements of x are all zero (and, for complex flavors, α is real), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau \leq 2$ (for real flavors), or

$1 \leq \text{Re}(\tau) \leq 2$ and $\text{abs}(\tau-1) \leq 1$ (for complex flavors).

Input Parameters

n	INTEGER. The order of the elementary reflector.
α	REAL for <code>slarfgp</code> DOUBLE PRECISION for <code>dlarfgp</code> COMPLEX for <code>clarfgp</code> DOUBLE COMPLEX for <code>zlarfgp</code> On entry, the value α .
x	REAL for <code>s</code> DOUBLE PRECISION for <code>dlarfgp</code> COMPLEX for <code>clarfgp</code> DOUBLE COMPLEX for <code>zlarfgp</code> Array, DIMENSION $(1+(n-2)*\text{abs}(\text{incx}))$. On entry, the vector x .
incx	INTEGER. The increment between elements of x . $\text{incx} > 0$.

Output Parameters

α	On exit, it is overwritten with the value β .
x	On exit, it is overwritten with the vector v .
τ	REAL for <code>slarfgp</code> DOUBLE PRECISION for <code>dlarfgp</code> COMPLEX for <code>clarfgp</code> DOUBLE COMPLEX for <code>zlarfgp</code>

The value τ .

?larft

Forms the triangular factor T of a block reflector $H = I - V^*T^*V^*H$.

Syntax

```
call slarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Include Files

- mkl.fi

Description

The routine ?larft forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If $direct = 'F'$, $H = H(1)^*H(2)^* \dots *H(k)$ and T is upper triangular;

If $direct = 'B'$, $H = H(k)^* \dots *H(2)^*H(1)$ and T is lower triangular.

If $storev = 'C'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the array v , and $H = I - V^*T^*V^T$ (for real flavors) or $H = I - V^*T^*V^H$ (for complex flavors) .

If $storev = 'R'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the array v , and $H = I - V^T^*T^*V$ (for real flavors) or $H = I - V^H^*T^*V$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

<i>direct</i>	CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: = 'F': $H = H(1)^*H(2)^* \dots *H(k)$ (forward) = 'B': $H = H(k)^* \dots *H(2)^*H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below): = 'C': column-wise = 'R': row-wise.
<i>n</i>	INTEGER. The order of the block reflector H . $n \geq 0$.
<i>k</i>	INTEGER. The order of the triangular factor T (equal to the number of elementary reflectors). $k \geq 1$.

<i>v</i>	<p>REAL for slarft</p> <p>DOUBLE PRECISION for dlarft</p> <p>COMPLEX for clarft</p> <p>DOUBLE COMPLEX for zlarft</p> <p>Array, DIMENSION</p> <p>(<i>ldv</i>, <i>k</i>) if <i>storev</i> = 'C' or</p> <p>(<i>ldv</i>, <i>n</i>) if <i>storev</i> = 'R'.</p> <p>The matrix <i>V</i>.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i>.</p> <p>If <i>storev</i> = 'C', $ldv \geq \max(1, n)$</p> <p>if <i>storev</i> = 'R', $ldv \geq k$.</p>
<i>tau</i>	<p>REAL for slarft</p> <p>DOUBLE PRECISION for dlarft</p> <p>COMPLEX for clarft</p> <p>DOUBLE COMPLEX for zlarft</p> <p>Array, size (<i>k</i>). <i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector <i>H</i>(<i>i</i>).</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of the output array <i>t</i>. $ldt \geq k$.</p>

Output Parameters

<i>t</i>	<p>REAL for slarft</p> <p>DOUBLE PRECISION for dlarft</p> <p>COMPLEX for clarft</p> <p>DOUBLE COMPLEX for zlarft</p> <p>Array, size <i>ldt</i> by <i>k</i>. The <i>k</i>-by-<i>k</i> triangular factor <i>T</i> of the block reflector. If <i>direct</i> = 'F', <i>T</i> is upper triangular; if <i>direct</i> = 'B', <i>T</i> is lower triangular. The rest of the array is not used.</p>
<i>v</i>	<p>The matrix <i>V</i>.</p>

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.



direct = 'B' and storev = 'C':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

direct = 'B' and storev = 'R':

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

Syntax

```
call slarfx( side, m, n, v, tau, c, ldc, work )
call dlarfx( side, m, n, v, tau, c, ldc, work )
call clarfx( side, m, n, v, tau, c, ldc, work )
call zlarfx( side, m, n, v, tau, c, ldc, work )
```

Include Files

- mkl.fi

Description

The routine ?larfx applies a real/complex elementary reflector *H* to a real/complex *m*-by-*n* matrix *C*, from either the left or the right.

H is represented in the following forms:

- $H = I - \tau v v^T$, where τ is a real scalar and v is a real vector.
- $H = I - \tau v v^H$, where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then *H* is taken to be the unit matrix.

Input Parameters

The data types are given for the Fortran interface.

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form H^*C If <i>side</i> = 'R': form C^*H .
<i>m</i>	INTEGER. The number of rows of the matrix <i>C</i> .
<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> .
<i>v</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx Array, size (<i>m</i>) if <i>side</i> = 'L' or (<i>n</i>) if <i>side</i> = 'R'. The vector <i>v</i> in the representation of <i>H</i> .
<i>tau</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx The value <i>tau</i> in the representation of <i>H</i> .
<i>c</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx Array, size <i>ldc</i> by <i>n</i> . On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $lda \geq (1,m)$.
<i>work</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx Workspace array, size (<i>n</i>) if <i>side</i> = 'L' or (<i>m</i>) if <i>side</i> = 'R'. <i>work</i> is not referenced if <i>H</i> has order < 11.

Output Parameters

c On exit, *c* is overwritten by the matrix H^*C if *side* = 'L', or C^*H if *side* = 'R'.

?large

Pre- and post-multiplies a real general matrix with a random orthogonal matrix.

Syntax

```
call slarge( n, a, lda, iseed, work, info )
```

```
call dlarge( n, a, lda, iseed, work, info )
```

```
call clarge( n, a, lda, iseed, work, info )
```

```
call zlarge( n, a, lda, iseed, work, info )
```

Include Files

- mkl.fi

Description

The routine ?large pre- and post-multiplies a general n -by- n matrix A with a random orthogonal or unitary matrix: $A = U^*D^*U^T$.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$
<i>a</i>	REAL for slarge, DOUBLE PRECISION for dlarge, COMPLEX for clarge, DOUBLE COMPLEX for zlarge, Array, size <i>lda</i> by n . On entry, the original n -by- n matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq n$.
<i>iseed</i>	INTEGER. Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and <i>iseed</i> (4) must be odd.
<i>work</i>	REAL for slarge, DOUBLE PRECISION for dlarge, COMPLEX for clarge, DOUBLE COMPLEX for zlarge, Workspace array, size $2*n$.

Output Parameters

<i>a</i>	INTEGER. On exit, <i>A</i> is overwritten by U^*A^*U' for some random orthogonal matrix <i>U</i> .
<i>iseed</i>	INTEGER. On exit, the seed is updated.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the <i>i</i> -th parameter had an illegal value.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv( n, x, incx, y, incy, c, incc )
call dlargv( n, x, incx, y, incy, c, incc )
call clargv( n, x, incx, y, incy, c, incc )
call zlargv( n, x, incx, y, incy, c, incc )
```

Include Files

- mkl.fi

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors *x* and *y*.

For *slargv*/*dlargv*:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

For *clargv*/*zlargv*:

$$\begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

where $c(i)^2 + \text{abs}(s(i))^2 = 1$ and the following conventions are used (these are the same as in *clartg*/*zlargv* but differ from the BLAS Level 1 routine *crotg*/*zrotg*):

If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;

If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n	INTEGER. The number of plane rotations to be generated.
x, y	REAL for slargv DOUBLE PRECISION for dlargv COMPLEX for clargv DOUBLE COMPLEX for zlargv Arrays, DIMENSION $(1+(n-1)*incx)$ and $(1+(n-1)*incy)$, respectively. On entry, the vectors x and y .
$incx$	INTEGER. The increment between elements of x . $incx > 0$.
$incy$	INTEGER. The increment between elements of y . $incy > 0$.
$incc$	INTEGER. The increment between elements of the output array c . $incc > 0$.

Output Parameters

x	On exit, $x(i)$ is overwritten by a_i (for real flavors), or by r_i (for complex flavors), for $i = 1, \dots, n$.
y	On exit, the sines $s(i)$ of the plane rotations.
c	REAL for slargv/clargv DOUBLE PRECISION for dlargv/zlargv Array, DIMENSION $(1+(n-1)*incc)$. The cosines of the plane rotations.

?larnd

Returns a random real number from a uniform or normal distribution.

Syntax

```
res = slarnd( idist, iseed )
res = dlarnd( idist, iseed )
res = clarnd( idist, iseed )
res = zlarnd( idist, iseed )
```

Include Files

- mkl.fi

Description

The routine ?larnd returns a random number from a uniform or normal distribution.

Input Parameters

<i>idist</i>	INTEGER. Specifies the distribution of the random numbers. For <code>slarnd</code> and <code>dlanrd</code> : = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1). For <code>clarnd</code> and <code>zlanrd</code> : = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc $\text{abs}(z) \leq 1$ = 5: uniformly distributed on the circle $\text{abs}(z) = 1$
<i>iseed</i>	INTEGER. Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and <code>iseed(4)</code> must be odd.

Output Parameters

<i>iseed</i>	INTEGER. On exit, the seed is updated.
<i>res</i>	REAL for <code>slarnd</code> , DOUBLE PRECISION for <code>dlanrd</code> , COMPLEX for <code>clarnd</code> , DOUBLE COMPLEX for <code>zlanrd</code> , Random number.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
call slarnv( idist, iseed, n, x )
call dlanrv( idist, iseed, n, x )
call clarnd( idist, iseed, n, x )
call zlanrv( idist, iseed, n, x )
```

Include Files

- `mkl.fi`

Description

The routine `?larnv` returns a vector of n random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine `?laruv` to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

The data types are given for the Fortran interface.

<i>idist</i>	INTEGER. Specifies the distribution of the random numbers: for <code>slarnv</code> and <code>dlarnv</code> : = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1). for <code>clarnv</code> and <code>zlarnv</code> : = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc $\text{abs}(z) < 1$ = 5: uniformly distributed on the circle $\text{abs}(z) = 1$
<i>iseed</i>	INTEGER. Array, size (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <code>iseed(4)</code> must be odd.
<i>n</i>	INTEGER. The number of random numbers to be generated.

Output Parameters

<i>x</i>	REAL for <code>slarnv</code> DOUBLE PRECISION for <code>dlarnv</code> COMPLEX for <code>clarnv</code> DOUBLE COMPLEX for <code>zlarnv</code> Array, size (n). The generated random numbers.
<i>iseed</i>	On exit, the seed is updated.

?laror

Pre- or post-multiplies an m -by- n matrix by a random orthogonal/unitary matrix.

Syntax

```
call slaror( side, init, m, n, a, lda, iseed, x, info )
call dlaror( side, init, m, n, a, lda, iseed, x, info )
call claror( side, init, m, n, a, lda, iseed, x, info )
```

call `zlaror(side, init, m, n, a, lda, iseed, x, info)`

Include Files

- `mkl.fi`

Description

The routine `?laror` pre- or post-multiplies an m -by- n matrix A by a random orthogonal or unitary matrix U , overwriting A . A may optionally be initialized to the identity matrix before multiplying by U . U is generated using the method of G.W. Stewart (SIAM J. Numer. Anal. 17, 1980, 403-409).

Input Parameters

<i>side</i>	<p>CHARACTER*1. Specifies whether A is multiplied by U on the left or right.</p> <p>for <code>slaror</code> and <code>dlaror</code>:</p> <p>If <i>side</i> = 'L', multiply A on the left (premultiply) by U.</p> <p>If <i>side</i> = 'R', multiply A on the right (postmultiply) by U^T.</p> <p>If <i>side</i> = 'C' or 'T', multiply A on the left by U and the right by U^T.</p> <p>for <code>claror</code> and <code>zlaror</code>:</p> <p>If <i>side</i> = 'L', multiply A on the left (premultiply) by U.</p> <p>If <i>side</i> = 'R', multiply A on the right (postmultiply) by $UC^>$.</p> <p>If <i>side</i> = 'C', multiply A on the left by U and the right by $UC^>$.</p> <p>If <i>side</i> = 'T', multiply A on the left by U and the right by U^T.</p>
<i>init</i>	<p>CHARACTER*1. Specifies whether or not a should be initialized to the identity matrix.</p> <p>If <i>init</i> = 'I', initialize a to (a section of) the identity matrix before applying U.</p> <p>If <i>init</i> = 'N', no initialization. Apply U to the input matrix A.</p> <p><i>init</i> = 'I' generates square or rectangular orthogonal matrices:</p> <p>For $m = n$ and <i>side</i> = 'L' or 'R', the rows and the columns are orthogonal to each other.</p> <p>For rectangular matrices where $m < n$:</p> <ul style="list-style-type: none"> • If <i>side</i> = 'R', <code>?laror</code> produces a dense matrix in which rows are orthogonal and columns are not. • If <i>side</i> = 'L', <code>?laror</code> produces a matrix in which rows are orthogonal, first m columns are orthogonal, and remaining columns are zero. <p>For rectangular matrices where $m > n$:</p> <ul style="list-style-type: none"> • If <i>side</i> = 'L', <code>?laror</code> produces a dense matrix in which columns are orthogonal and rows are not. • If <i>side</i> = 'R', <code>?laror</code> produces a matrix in which columns are orthogonal, first m rows are orthogonal, and remaining rows are zero.
<i>m</i>	INTEGER. The number of rows of A .
<i>n</i>	INTEGER. The number of columns of A .

a REAL for slaror,
DOUBLE PRECISION for dlaror,
COMPLEX for claror,
DOUBLE COMPLEX for zlaror,
Array, size *lda* by *n*.

lda INTEGER. The leading dimension of the array *a*.
 $lda \geq \max(1, m)$.

iseed INTEGER. Array, size (4).
On entry, specifies the seed of the random number generator. The array elements must be between 0 and 4095; if not they are reduced mod 4096. Also, *iseed*(4) must be odd.

x REAL for slaror,
DOUBLE PRECISION for dlaror,
COMPLEX for claror,
DOUBLE COMPLEX for zlaror,
Workspace array, size $(3 * \max(m, n))$.

Value of <i>side</i>	Length of workspace
'L'	$2 * m + n$
'R'	$2 * n + m$
'C' or 'T'	$3 * n$

Output Parameters

a On exit, overwritten
by *UA* (if *side* = 'L'),
by *AU* (if *side* = 'R'),
by *UAU^T* (if *side* = 'C' or 'T').

iseed The values of *iseed* are changed on exit, and can be used in the next call to continue the same random number sequence.

info INTEGER. Array, size (4).
For slaror and dlaror:
If *info* = 0, the execution is successful.
If *info* < 0, the *i*-th parameter had an illegal value.
If *info* = 1, the random numbers generated by ?laror are bad.
For claror and zlaror:
If *info* = 0, the execution is successful.
If *info* = -1, *side* is not 'L', 'R', 'C', or 'T'.
If *info* = -3, if *m* is negative.

If *info* = -4, if *m* is negative or if *side* is 'C' or 'T' and *n* is not equal to *m* .

If *info* = -6, if *lda* is less than *m* .

?larot

Applies a Givens rotation to two adjacent rows or columns.

Syntax

```
call slarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
```

```
call dlarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
```

```
call clarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
```

```
call zlarot( lrows, lleft, lright, nl, c, s, a, lda, xleft, xright )
```

Include Files

- mkl.fi

Description

The routine ?larot applies a Givens rotation to two adjacent rows or columns, where one element of the first or last column or row is stored in some format other than GE so that elements of the matrix may be used or modified for which no array element is provided.

One example is a symmetric matrix in SB format (bandwidth = 4), for which *uplo* = 'L'. Two adjacent rows will have the format:

```
row j :      C > C > C > C > C > . . . .
row j + 1 :  C > C > C > C > C > . . . .
```

'*' indicates elements for which storage is provided.

'.' indicates elements for which no storage is provided, but are not necessarily zero; their values are determined by symmetry.

' ' indicates elements which are required to be zero, and have no storage provided.

Those columns which have two '*' entries can be handled by *srot* (for *slarot* and *clarot*), or by *drot* (for *dlarot* and *zlarot*).

Those columns which have no '*' entries can be ignored, since as long as the Givens rotations are carefully applied to preserve symmetry, their values are determined.

Those columns which have one '*' have to be handled separately, by using separate variables *p* and *q* :

```
row j :      C > C > C > C > C > p . . . .
row j + 1 :  q C > C > C > C > C > . . . .
```

If element *p* is set correctly, ?larot rotates the column and sets *p* to its new value. The next call to ?larot rotates columns *j* and *j + 1*, and restore symmetry. The element *q* is zero at the beginning, and non-zero after the rotation. Later, rotations would presumably be chosen to zero *q* out.

Typical Calling Sequences: rotating the *i*-th and (*i + 1*)-st rows.

Example

Typical Calling Sequences

These examples rotate the *i*-th and (*i + 1*)-st rows.

General dense matrix:

```
call dlarot (.TRUE.,.FALSE.,.FALSE., n, c, s,
a(i,1),lda, dummy, dummy)
```

General banded matrix in GB format:

```
j = max(1, i-kl )
nl = min( n, i+ku+1 ) + 1-j
call dlarot( .TRUE., i-kl.GE.1, i+ku.LT.n, nl, c,s,
a(ku+i+1-j,j),lda-1, xleft, xright )
```

NOTE

$i + 1 - j$ is just $\min(i, kl + 1)$.

Symmetric banded matrix in SY format, bandwidth K, lower triangle only:

```
j = max(1, i-k )
nl = min( k+1, i ) + 1
call dlarot( .TRUE., i-k.GE.1, .TRUE., nl, c,s,
a(i,j), lda, xleft, xright )
```

Same, but upper triangle only:

```
nl = min( k+1, n-i ) + 1
call dlarot( .TRUE., .TRUE., i+k.LT.n, nl, c,s,
a(i,i), lda, xleft, xright )
```

Symmetric banded matrix in SB format, bandwidth K, lower triangle only: [same as for SY, except:]

```
. . . . .
a(i+1-j,j), lda, xleft, xright )
```

NOTE

$i+1-j$ is just $\min(i,k+1)$

Same, but upper triangle only:

```
. . . . .
a(k+1,i), lda-1, xleft, xright )
```

Rotating columns is just the transpose of rotating rows, except for GB and SB: (rotating columns i and $i+1$)
GB:

NOTE

$ku+j+1-i$ is just $\max(1,ku+2-i)$

```
j = max(1, i-ku )
nl = min( n, i+kl+1 ) + 1-j
call dlarot( .TRUE., i-ku.LE.1, i+kl.LT.n, nl, c,s,
a(ku+j+1-i,i),lda-1, xtop, xbottm )
```

SB: (upper triangle)

```
. . . . .
a(k+j+1-i,i),lda-1, xtop, xbottm )
```

SB: (lower triangle) A(1,i),LDA-1, XTOP, XBOTTOM)

```
. . . . .
a(1,i),lda-1, xtop, xbottm )
```

Input Parameters

<i>lrows</i>	<p>LOGICAL.</p> <p>If <i>lrows</i> = <code>.TRUE.</code>, <code>?larot</code> rotates two rows.</p> <p>If <i>lrows</i> = <code>.FALSE.</code>, <code>?larot</code> rotates two columns.</p>
<i>lleft</i>	<p>LOGICAL.</p> <p>If <i>lleft</i> = <code>.TRUE.</code>, <i>xleft</i> is used instead of the corresponding element of <i>a</i> for the first element in the second row (if <i>lrows</i> = <code>.FALSE.</code>) or column (if <i>lrows</i>=<code>.TRUE.</code>).</p> <p>If <i>lleft</i> = <code>.FALSE.</code>, the corresponding element of <i>a</i> is used.</p>
<i>lright</i>	<p>LOGICAL.</p> <p>If <i>lleft</i> = <code>.TRUE.</code>, <i>xright</i> is used instead of the corresponding element of <i>a</i> for the first element in the second row (if <i>lrows</i> = <code>.FALSE.</code>) or column (if <i>lrows</i>=<code>.TRUE.</code>).</p> <p>If <i>lright</i> = <code>.FALSE.</code>, the corresponding element of <i>a</i> is used.</p>
<i>nl</i>	<p>INTEGER. The length of the rows (if <i>lrows</i>=<code>.TRUE.</code>) or columns (if <i>lrows</i>=<code>.TRUE.</code>) to be rotated.</p> <p>If <i>xleft</i> or <i>xright</i> are used, the columns or rows they are in should be included in <i>nl</i>, e.g., if <i>lleft</i> = <i>lright</i> = <code>.TRUE.</code>, then <i>nl</i> must be at least 2.</p> <p>The number of rows or columns to be rotated exclusive of those involving <i>xleft</i> and/or <i>xright</i> may not be negative, i.e., <i>nl</i> minus how many of <i>lleft</i> and <i>lright</i> are <code>.TRUE.</code> must be at least zero; if not, <code>xerbla</code> is called.</p>
<i>c, s</i>	<p>REAL for <code>slarot</code>,</p> <p>DOUBLE PRECISION for <code>dlarot</code>,</p> <p>COMPLEX for <code>clarot</code>,</p> <p>DOUBLE COMPLEX for <code>zlarot</code>,</p> <p>Specify the Givens rotation to be applied.</p> <p>If <i>lrows</i> = <code>.TRUE.</code>, then the matrix</p> $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ <p>is applied from the left.</p> <p>If <i>lrows</i> = <code>.FALSE.</code>, then the transpose thereof is applied from the right.</p>
<i>a</i>	<p>REAL for <code>slarot</code>,</p> <p>DOUBLE PRECISION for <code>dlarot</code>,</p> <p>COMPLEX for <code>clarot</code>,</p> <p>DOUBLE COMPLEX for <code>zlarot</code>,</p>

The array containing the rows or columns to be rotated. The first element of *a* should be the upper left element to be rotated.

lda

INTEGER. The "effective" leading dimension of *a*.

If *a* contains a matrix stored in GE or SY format, then this is just the leading dimension of *A*.

If *a* contains a matrix stored in band (GB or SB) format, then this should be one less than the leading dimension used in the calling routine. Thus, if *a* is dimensioned *a(lda,*)* in *?larot*, then *a(1, j)* would be the *j*-th element in the first of the two rows to be rotated, and *a(2, j)* would be the *j*-th in the second, regardless of how the array may be stored in the calling routine. *a* cannot be dimensioned, because for band format the row number may exceed *lda*, which is not legal FORTRAN.

If *lrows* = *.TRUE.*, then *lda* must be at least 1, otherwise it must be at least *nl* minus the number of *.TRUE.* values in *xleft* and *xright*.

xleft

REAL for *slarot*,

DOUBLE PRECISION for *dlarot*,

COMPLEX for *clarot*,

DOUBLE COMPLEX for *zlarot*,

If *lrows* = *.TRUE.*, *xleft* is used and modified instead of *a(2,1)* (if *lrows* = *.TRUE.*) or *a(1,2)* (if *lrows* = *.FALSE.*).

xright

REAL for *slarot*,

DOUBLE PRECISION for *dlarot*,

COMPLEX for *clarot*,

DOUBLE COMPLEX for *zlarot*,

If *lright* = *.TRUE.*, *xright* is used and modified instead of *a(1,nl)* (if *lrows* = *.TRUE.*) or *a(nl,1)* (if *lrows* = *.FALSE.*).

Output Parameters

a

On exit, modified array *A*.

?larra

Computes the splitting points with the specified threshold.

Syntax

```
call slarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
```

```
call dlarra( n, d, e, e2, spltol, tnrm, nsplit, isplit, info )
```

Include Files

- `mkl.fi`

Description

The routine computes the splitting points with the specified threshold and sets any "small" off-diagonal elements to zero.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix ($n > 1$).
<i>d</i>	REAL for <i>slarra</i> DOUBLE PRECISION for <i>dlarra</i> Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> .
<i>e</i>	REAL for <i>slarra</i> DOUBLE PRECISION for <i>dlarra</i> Array, DIMENSION (<i>n</i>). First (<i>n</i> -1) entries contain the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e</i> (<i>n</i>) need not be set.
<i>e2</i>	REAL for <i>slarra</i> DOUBLE PRECISION for <i>dlarra</i> Array, DIMENSION (<i>n</i>). First (<i>n</i> -1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e2</i> (<i>n</i>) need not be set.
<i>spltol</i>	REAL for <i>slarra</i> DOUBLE PRECISION for <i>dlarra</i> The threshold for splitting. Two criteria can be used: <i>spltol</i> <0 : criterion based on absolute off-diagonal value; <i>spltol</i> >0 : criterion that preserves relative accuracy.
<i>tnrm</i>	REAL for <i>slarra</i> DOUBLE PRECISION for <i>dlarra</i> The norm of the matrix.

Output Parameters

<i>e</i>	On exit, the entries <i>e</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, are set to zero, the other entries of <i>e</i> are untouched.
<i>e2</i>	On exit, the entries <i>e2</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, are set to zero.
<i>nsplit</i>	INTEGER. The number of blocks the matrix <i>T</i> splits into. $1 \leq nsplit \leq n$
<i>isplit</i>	INTEGER. Array, DIMENSION (<i>n</i>).

The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, and so on, and the $nsplit$ -th consists of rows/columns $isplit(nsplitt-1)+1$ through $isplit(nsplitt)=n$.

info INTEGER.
= 0: successful exit.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
            iwork, pivmin, spdiam, twist, info )
```

```
call dlarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
            iwork, pivmin, spdiam, twist, info )
```

Include Files

- mkl.fi

Description

Given the relatively robust representation (RRR) $L^*D^*L^T$, the routine does "limited" bisection to refine the eigenvalues of $L^*D^*L^T$, $w(ifirst-offset)$ through $w(ilast-offset)$, to more accuracy. Initial guesses for these eigenvalues are input in w . The corresponding estimate of the error in these guesses and their gaps are input in $werr$ and $wgap$, respectively. During bisection, intervals $[left, right]$ are maintained by storing their mid-points and semi-widths in the arrays w and $werr$ respectively.

Input Parameters

n INTEGER. The order of the matrix.

d REAL for slarrb
DOUBLE PRECISION for dlarrb
Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D .

lld REAL for slarrb
DOUBLE PRECISION for dlarrb
Array, DIMENSION ($n-1$).
The $n-1$ elements $L_i^*L_i^*D_i$.

ifirst INTEGER. The index of the first eigenvalue to be computed.

ilast INTEGER. The index of the last eigenvalue to be computed.

rtol1, rtol2 REAL for slarrb
DOUBLE PRECISION for dlarrb

Tolerance for the convergence of the bisection intervals. An interval $[left, right]$ has converged if $RIGHT-LEFT.LT.MAX(rtol1*gap, rtol2*max(|left|, |right|))$, where gap is the (estimated) distance to the nearest eigenvalue.

offset

INTEGER. Offset for the arrays w , $wgap$ and $werr$, that is, the *ifirst-offset* through *ilast-offset* elements of these arrays are to be used.

w

REAL for slarrb

DOUBLE PRECISION for dlarrb

Array, DIMENSION (n). On input, $w(ifirst-offset)$ through $w(ilast-offset)$ are estimates of the eigenvalues of $L*D*L^T$ indexed *ifirst* through *ilast*.

wgap

REAL for slarrb

DOUBLE PRECISION for dlarrb

Array, DIMENSION ($n-1$). The estimated gaps between consecutive eigenvalues of $L*D*L^T$, that is, $wgap(i-offset)$ is the gap between eigenvalues i and $i+1$. Note that if $IFIRST.EQ.ILAST$ then $wgap(ifirst-offset)$ must be set to 0.

werr

REAL for slarrb

DOUBLE PRECISION for dlarrb

Array, DIMENSION (n). On input, $werr(ifirst-offset)$ through $werr(ilast-offset)$ are the errors in the estimates of the corresponding elements in w .

work

REAL for slarrb

DOUBLE PRECISION for dlarrb

Workspace array, DIMENSION ($2*n$).

pivmin

REAL for slarrb

DOUBLE PRECISION for dlarrb

The minimum pivot in the Sturm sequence.

spdiam

REAL for slarrb

DOUBLE PRECISION for dlarrb

The spectral diameter of the matrix.

twist

INTEGER. The twist index for the twisted factorization that is used for the *negcount*.

$$twist = n: \text{ Compute negcount from } L*D*L^T - \lambda_i = L+* D+ *L+^T$$

$$twist = n: \text{ Compute negcount from } L*D*L^T - \lambda_i = U-*D-*U^-T$$

$$twist = n: \text{ Compute negcount from } L*D*L^T - \lambda_i = N_r*D_r*N_r$$

iwork

INTEGER.

Workspace array, DIMENSION ($2*n$).

Output Parameters

<i>w</i>	On output, the estimates of the eigenvalues are "refined".
<i>wgap</i>	On output, the gaps are refined.
<i>werr</i>	On output, "refined" errors in the estimates of <i>w</i> .
<i>info</i>	INTEGER. Error flag.

?larrc

Computes the number of eigenvalues of the symmetric tridiagonal matrix.

Syntax

```
call slarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
call dlarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
```

Include Files

- mkl.fi

Description

The routine finds the number of eigenvalues of the symmetric tridiagonal matrix T or of its factorization L^*D*L^T in the specified interval.

Input Parameters

<i>jobt</i>	CHARACTER*1. = 'T': computes Sturm count for matrix T . = 'L': computes Sturm count for matrix L^*D*L^T .
<i>n</i>	INTEGER. The order of the matrix. ($n > 1$).
<i>vl,vu</i>	REAL for slarrc DOUBLE PRECISION for dlarrc The lower and upper bounds for the eigenvalues.
<i>d</i>	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (n). If <i>jobt</i> = 'T': contains the n diagonal elements of the tridiagonal matrix T . If <i>jobt</i> = 'L': contains the n diagonal elements of the diagonal matrix D .
<i>e</i>	REAL for slarrc DOUBLE PRECISION for dlarrc

Array, DIMENSION (n).

If $jobt= 'T'$: contains the $(n-1)$ offdiagonal elements of the matrix T .

If $jobt= 'L'$: contains the $(n-1)$ offdiagonal elements of the matrix L .

pivmin

REAL for slarrc

DOUBLE PRECISION for dlarrc

The minimum pivot in the Sturm sequence for the matrix T .

Output Parameters

eigcnt

INTEGER.

The number of eigenvalues of the symmetric tridiagonal matrix T that are in the half-open interval $(vl, vu]$.

lcnt,rcnt

INTEGER.

The left and right negcounts of the interval.

info

INTEGER.

Now it is not used and always is set to 0.

?larrd

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

```
call dlarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine computes the eigenvalues of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from ?stemr. The user may ask for all eigenvalues, all eigenvalues in the half-open interval $(vl, vu]$, or the il -th through iu -th eigenvalues.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66]).

Input Parameters

range

CHARACTER.

= 'A': ("All") all eigenvalues will be found.

	<p>= 'V': ("Value") all eigenvalues in the half-open interval (v_l, v_u] will be found.</p> <p>= 'I': ("Index") the i_l-th through i_u-th eigenvalues will be found.</p>
<i>order</i>	<p>CHARACTER.</p> <p>= 'B': ("By block") the eigenvalues will be grouped by split-off block (see <i>iblock</i>, <i>isplit</i> below) and ordered from smallest to largest within the block.</p> <p>= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.</p>
<i>n</i>	<p>INTEGER. The order of the tridiagonal matrix T ($n \geq 1$).</p>
<i>v_l, v_u</i>	<p>REAL for <i>slarrd</i></p> <p>DOUBLE PRECISION for <i>dlarrd</i></p> <p>If <i>range</i> = 'V': the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to v_l, or greater than v_u, will not be returned. $v_l < v_u$.</p> <p>If <i>range</i> = 'A' or 'I': not referenced.</p>
<i>i_l, i_u</i>	<p>INTEGER.</p> <p>If <i>range</i> = 'I': the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq i_l \leq i_u \leq n$, if $n > 0$; $i_l=1$ and $i_u=0$ if $n=0$.</p> <p>If <i>range</i> = 'A' or 'V': not referenced.</p>
<i>gers</i>	<p>REAL for <i>slarrd</i></p> <p>DOUBLE PRECISION for <i>dlarrd</i></p> <p>Array, DIMENSION ($2*n$).</p> <p>The n Gerschgorin intervals (the i-th Gerschgorin interval is ($gers(2*i-1)$, $gers(2*i)$).</p>
<i>reltol</i>	<p>REAL for <i>slarrd</i></p> <p>DOUBLE PRECISION for <i>dlarrd</i></p> <p>The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least <i>radix*machine epsilon</i>.</p>
<i>d</i>	<p>REAL for <i>slarrd</i></p> <p>DOUBLE PRECISION for <i>dlarrd</i></p> <p>Array, DIMENSION (n).</p> <p>Contains n diagonal elements of the tridiagonal matrix T.</p>
<i>e</i>	<p>REAL for <i>slarrd</i></p> <p>DOUBLE PRECISION for <i>dlarrd</i></p>

	<p>Array, DIMENSION ($n-1$).</p> <p>Contains ($n-1$) off-diagonal elements of the tridiagonal matrix T.</p>
<i>e2</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>Array, DIMENSION ($n-1$).</p> <p>Contains ($n-1$) squared off-diagonal elements of the tridiagonal matrix T.</p>
<i>pivmin</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>The minimum pivot in the Sturm sequence for the matrix T.</p>
<i>nsplit</i>	<p>INTEGER.</p> <p>The number of diagonal blocks the matrix T. $1 \leq nsplit \leq n$</p>
<i>isplit</i>	<p>INTEGER.</p> <p>Arrays, DIMENSION (n).</p> <p>The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, and so on, and the $nsplit$-th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.</p> <p>(Only the first $nsplit$ elements actually is used, but since the user cannot know a priori value of $nsplit$, n words must be reserved for $isplit$.)</p>
<i>work</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>Workspace array, DIMENSION ($4*n$).</p>
<i>iwork</i>	<p>INTEGER.</p> <p>Workspace array, DIMENSION ($4*n$).</p>

Output Parameters

<i>m</i>	<p>INTEGER.</p> <p>The actual number of eigenvalues found. $0 \leq m \leq n$. (See also the description of <code>info=2,3</code>.)</p>
<i>w</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>Array, DIMENSION (n).</p> <p>The first m elements of w contain the eigenvalue approximations. <code>?laprd</code> computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j. The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j) / 2$. The corresponding error is bounded by $werr(j) = \text{abs}(a_j - b_j) / 2$.</p>

<i>werr</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>Array, DIMENSION (<i>n</i>).</p> <p>The error bound on the corresponding eigenvalue approximation in <i>w</i>.</p>
<i>wl, wu</i>	<p>REAL for <code>slarrd</code></p> <p>DOUBLE PRECISION for <code>dlarrd</code></p> <p>The interval (<i>wl</i>, <i>wu</i>] contains all the wanted eigenvalues.</p> <p>If <i>range</i> = 'V': then <i>wl</i>=<i>vl</i> and <i>wu</i>=<i>vu</i>.</p> <p>If <i>range</i> = 'A': then <i>wl</i> and <i>wu</i> are the global Gerschgorin bounds on the spectrum.</p> <p>If <i>range</i> = 'I': then <i>wl</i> and <i>wu</i> are computed by <code>?laebz</code> from the index range specified.</p>
<i>iblock</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>At each row/column <i>j</i> where <i>e</i>(<i>j</i>) is zero or small, the matrix <i>T</i> is considered to split into a block diagonal matrix.</p> <p>If <i>info</i> = 0, then <i>iblock</i>(<i>i</i>) specifies to which block (from 1 to the number of blocks) the eigenvalue <i>w</i>(<i>i</i>) belongs. (The routine may use the remaining <i>n-m</i> elements as workspace.)</p>
<i>indexw</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>The indices of the eigenvalues within each block (submatrix); for example, <i>indexw</i>(<i>i</i>) = <i>j</i> and <i>iblock</i>(<i>i</i>) = <i>k</i> imply that the <i>i</i>-th eigenvalue <i>w</i>(<i>i</i>) is the <i>j</i>-th eigenvalue in block <i>k</i>.</p>
<i>info</i>	<p>INTEGER.</p> <p>= 0: successful exit.</p> <p>< 0: if <i>info</i> = -<i>i</i>, the <i>i</i>-th argument has an illegal value</p> <p>> 0: some or all of the eigenvalues fail to converge or are not computed:</p> <p>=1 or 3: bisection fail to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.</p> <p>=2 or 3: <i>range</i>='I' only: not all of the eigenvalues <i>il</i>:<i>iu</i> are found.</p> <p>=4: <i>range</i>='I', and the Gerschgorin interval initially used is too small. No eigenvalues are computed.</p>

?larre

Given the tridiagonal matrix *T*, sets small off-diagonal elements to zero and for each unreduced block *T_i*, finds base representations and eigenvalues.

Syntax

```
call slarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

```
call dlarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Include Files

- mkl.fi

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , the routine sets any "small" off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block spectrum
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then used by `?stemr` to compute the eigenvectors of a symmetric tridiagonal matrix. The accuracy varies depending on whether bisection is used to find a few eigenvalues or the `dqds` algorithm (subroutine `?lasq2`) to compute all and discard any unwanted one. As an added benefit, `?larre` also outputs the n Gerschgorin intervals for the matrices $L_i D_i L_i^T$.

Input Parameters

<i>range</i>	CHARACTER. = 'A': ("All") all eigenvalues will be found. = 'V': ("Value") all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found. = 'I': ("Index") the <i>il</i> -th through <i>iu</i> -th eigenvalues of the entire matrix will be found.
<i>n</i>	INTEGER. The order of the matrix. $n > 0$.
<i>vl, vu</i>	REAL for <code>slarre</code> DOUBLE PRECISION for <code>dlarre</code> If <i>range</i> ='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <i>vl</i> , or greater than <i>vu</i> , are not returned. $vl < vu$.
<i>il, iu</i>	INTEGER. If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$.
<i>d</i>	REAL for <code>slarre</code> DOUBLE PRECISION for <code>dlarre</code> Array, DIMENSION (<i>n</i>). The n diagonal elements of the diagonal matrices T .
<i>e</i>	REAL for <code>slarre</code>

	DOUBLE PRECISION for dlarre
	Array, DIMENSION (n). The first ($n-1$) entries contain the subdiagonal elements of the tridiagonal matrix T ; $e(n)$ need not be set.
$e2$	REAL for slarre DOUBLE PRECISION for dlarre
	Array, DIMENSION (n). The first ($n-1$) entries contain the squares of the subdiagonal elements of the tridiagonal matrix T ; $e2(n)$ need not be set.
$rtol1, rtol2$	REAL for slarre DOUBLE PRECISION for dlarre
	Parameters for bisection. An interval $[LEFT, RIGHT]$ has converged if $RIGHT-LEFT.LT.MAX(rtol1*gap, rtol2*max(LEFT , RIGHT))$.
$spltol$	REAL for slarre DOUBLE PRECISION for dlarre
	The threshold for splitting.
$work$	REAL for slarre DOUBLE PRECISION for dlarre
	Workspace array, DIMENSION ($6*n$).
$iwork$	INTEGER. Workspace array, DIMENSION ($5*n$).

Output Parameters

vl, vu	On exit, if $range = 'I'$ or $'A'$, contain the bounds on the desired part of the spectrum.
d	On exit, the n diagonal elements of the diagonal matrices D_i .
e	On exit, the subdiagonal elements of the unit bidiagonal matrices L_i . The entries $e(isplit(i))$, $1 \leq i \leq nsplit$, contain the base points $sigma_i$ on output.
$e2$	On exit, the entries $e2(isplit(i))$, $1 \leq i \leq nsplit$, have been set to zero.
$nsplit$	INTEGER. The number of blocks T splits into. $1 \leq nsplit \leq n$.
$isplit$	INTEGER. Array, DIMENSION (n). The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, etc., and the $nsplit$ -th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.
m	INTEGER. The total number of eigenvalues (of all the $L_i * D_i * L_i^T$) found.
w	REAL for slarre DOUBLE PRECISION for dlarre

Array, DIMENSION (n). The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i * D_i * L_i^T$, are sorted in ascending order. The routine may use the remaining $n-m$ elements as workspace.

werr

REAL for *slarre*

DOUBLE PRECISION for *dlarre*

Array, DIMENSION (n). The error bound on the corresponding eigenvalue in w .

wgap

REAL for *slarre*

DOUBLE PRECISION for *dlarre*

Array, DIMENSION (n). The separation from the right neighbor eigenvalue in w . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block the left gap is stored.

iblock

INTEGER. Array, DIMENSION (n).

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $=2$ if $w(i)$ belongs to the second block, etc.

indexw

INTEGER. Array, DIMENSION (n).

The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers

REAL for *slarre*

DOUBLE PRECISION for *dlarre*

Array, DIMENSION ($2*n$). The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).

pivmin

REAL for *slarre*

DOUBLE PRECISION for *dlarre*

The minimum pivot in the Sturm sequence for T .

info

INTEGER.

If $info = 0$: successful exit

If $info > 0$: A problem occurred in *?larre*. If $info = 5$, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If $info < 0$: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter *info* for further information is required.

- If $info = -1$, there is a problem in *?larrd*
- If $info = -2$, no base representation could be found in *maxtry* iterations. Increasing *maxtry* and recompilation might be a remedy.
- If $info = -3$, there is a problem in *?larrb* when computing the refined root representation for *?lasq2*.

- If $info = -4$, there is a problem in `?larrb` when performing bisection on the desired part of the spectrum.
- If $info = -5$, there is a problem in `?lasq2`.
- If $info = -6$, there is a problem in `?lasq2`.

See Also

`?stemr`
`?lasq2`
`?larrb`
`?larrd`

?larrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr,
pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf( n, d, l, ld, clstrt, clend, w, wgap, werr, spdiam, clgapl, clgapr,
pivmin, sigma, dplus, lplus, work, info )
```

Include Files

- `mkl.fi`

Description

Given the initial representation $L*D*L^T$ and its cluster of close eigenvalues (in a relative measure), $w(clstrt)$, $w(clstrt+1)$, ... $w(clend)$, the routine `?larrf` finds a new relatively robust representation

$$L*D*L^T - \sigma_i * I = L(+)*D(+)*L(+)^T$$

such that at least one of the eigenvalues of $L(+)*D(+)*L(+)^T$ is relatively isolated.

Input Parameters

n	INTEGER. The order of the matrix (subblock, if the matrix is splitted).
d	REAL for <code>slarrf</code> DOUBLE PRECISION for <code>dlarrf</code> Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D .
l	REAL for <code>slarrf</code> DOUBLE PRECISION for <code>dlarrf</code> Array, DIMENSION ($n-1$). The ($n-1$) subdiagonal elements of the unit bidiagonal matrix L .
ld	REAL for <code>slarrf</code> DOUBLE PRECISION for <code>dlarrf</code> Array, DIMENSION ($n-1$).

The $n-1$ elements $L_i * D_i$.

<i>clstrt</i>	INTEGER. The index of the first eigenvalue in the cluster.
<i>clend</i>	INTEGER. The index of the last eigenvalue in the cluster.
<i>w</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The eigenvalue approximations of $L * D * L^T$ in ascending order. $w(clstrt)$ through $w(clend)$ form the cluster of relatively close eigenvalues.
<i>wgap</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The separation from the right neighbor eigenvalue in w .
<i>werr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. On input, <i>werr</i> contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in w .
<i>spdiam</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Estimate of the spectral diameter obtained from the Gerschgorin intervals.
<i>clgapl, clgapr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.
<i>pivmin</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The minimum pivot allowed in the Sturm sequence.
<i>work</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Workspace array, DIMENSION $(2 * n)$.

Output Parameters

<i>wgap</i>	On output, the gaps are refined.
<i>sigma</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The shift used to form $L(+)*D*(+)*L(+)^T$.

<i>dplus</i>	REAL for <i>slarrf</i> DOUBLE PRECISION for <i>dlarrf</i> Array, DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> (+).
<i>lplus</i>	REAL for <i>slarrf</i> DOUBLE PRECISION for <i>dlarrf</i> Array, DIMENSION (<i>n</i>). The first (<i>n</i> -1) elements of <i>lplus</i> contain the subdiagonal elements of the unit bidiagonal matrix <i>L</i> (+).

?larrj

Performs refinement of the initial estimates of the eigenvalues of the matrix T.

Syntax

```
call slarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork, pivmin,
           spdiam, info )
```

```
call dlarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork, pivmin,
           spdiam, info )
```

Include Files

- `mkl.fi`

Description

Given the initial eigenvalue approximations of *T*, this routine does bisection to refine the eigenvalues of *T*, *w*(*ifirst*-*offset*) through *w*(*ilast*-*offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses in *werr*. During bisection, intervals [*a*,*b*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix <i>T</i> .
<i>d</i>	REAL for <i>slarrj</i> DOUBLE PRECISION for <i>dlarrj</i> Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the matrix <i>T</i> .
<i>e2</i>	REAL for <i>slarrj</i> DOUBLE PRECISION for <i>dlarrj</i> Array, DIMENSION (<i>n</i> -1). Contains (<i>n</i> -1) squared sub-diagonal elements of the <i>T</i> .
<i>ifirst</i>	INTEGER. The index of the first eigenvalue to be computed.

<i>ilast</i>	INTEGER. The index of the last eigenvalue to be computed.
<i>rtol</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Tolerance for the convergence of the bisection intervals. An interval [a,b] is considered to be converged if $(b-a) \leq rtol * \max(a , b)$.
<i>offset</i>	INTEGER. Offset for the arrays <i>w</i> and <i>werr</i> , that is the <i>ifirst-offset</i> through <i>ilast-offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (<i>n</i>). On input, <i>w(ifirst-offset)</i> through <i>w(ilast-offset)</i> are estimates of the eigenvalues of L^*D*L^T indexed <i>ifirst</i> through <i>ilast</i> .
<i>werr</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (<i>n</i>). On input, <i>werr(ifirst-offset)</i> through <i>werr(ilast-offset)</i> are the errors in the estimates of the corresponding elements in <i>w</i> .
<i>work</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Workspace array, DIMENSION ($2*n$).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION ($2*n$).
<i>pivmin</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The minimum pivot in the Sturm sequence for the matrix <i>T</i> .
<i>spdiam</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The spectral diameter of the matrix <i>T</i> .

Output Parameters

<i>w</i>	On exit, contains the refined estimates of the eigenvalues.
<i>werr</i>	On exit, contains the refined errors in the estimates of the corresponding elements in <i>w</i> .
<i>info</i>	INTEGER.

Now it is not used and always is set to 0.

?larrk

Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.

Syntax

```
call slarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
call dlarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
```

Include Files

- mkl.fi

Description

The routine computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from ?stemr.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. For more details see [[Kahan66](#)].

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T . ($n \geq 1$).
<i>iw</i>	INTEGER. The index of the eigenvalue to be returned.
<i>gl, gu</i>	REAL for slarrk DOUBLE PRECISION for dlarrk An upper and a lower bound on the eigenvalue.
<i>d</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION (n). Contains n diagonal elements of the matrix T .
<i>e2</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION ($n-1$). Contains $(n-1)$ squared off-diagonal elements of the T .
<i>pivmin</i>	REAL for slarrk DOUBLE PRECISION for dlarrk The minimum pivot in the Sturm sequence for the matrix T .
<i>reltol</i>	REAL for slarrk

DOUBLE PRECISION for dlarrk

The minimum relative width of an interval. When an interval is narrower than *reltol* times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least *radix*machine epsilon*.

Output Parameters

<i>w</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the eigenvalue approximation.
<i>werr</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the error bound on the corresponding eigenvalue approximation in <i>w</i> .
<i>info</i>	INTEGER. = 0: Eigenvalue converges = -1: Eigenvalue does not converge

?larr

Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Syntax

call slarrr(*n*, *d*, *e*, *info*)

call dlarrr(*n*, *d*, *e*, *info*)

Include Files

- mkl.fi

Description

The routine performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T . ($n > 0$).
<i>d</i>	REAL for slarrr DOUBLE PRECISION for dlarrr Array, DIMENSION (n). Contains n diagonal elements of the matrix T .

`e` REAL for `slarrv`
 DOUBLE PRECISION for `dlarrv`
 Array, DIMENSION (n).
 The first $(n-1)$ entries contain sub-diagonal elements of the tridiagonal matrix T ; $e(n)$ is set to 0.

Output Parameters

`info` INTEGER.
 = 0: the matrix warrants computations preserving relative accuracy (default value).
 = -1: the matrix warrants computations guaranteeing only absolute accuracy.

?larrv

Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L , D and the eigenvalues of $L*D*L^T$.

Syntax

```
call slarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

```
call dlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

```
call clarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

```
call zlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtol1, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

Include Files

- `mkl.fi`

Description

The routine `?larrv` computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L , D and approximations to the eigenvalues of $L*D*L^T$.

The input eigenvalues should have been computed by `slarre` for real flavors (`slarrv/clarrv`) and by `dlarre` for double precision flavors (`dlarrv/zlarrv`).

Input Parameters

`n` INTEGER. The order of the matrix. $n \geq 0$.

`vl, vu` REAL for `slarrv/clarrv`

DOUBLE PRECISION for `dlarrv/zlarrv`

Lower and upper bounds respectively of the interval that contains the desired eigenvalues. $v_l < v_u$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range.

d REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 Array, DIMENSION (*n*). On entry, the *n* diagonal elements of the diagonal matrix *D*.

l REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 Array, DIMENSION (*n*).
 On entry, the (*n*-1) subdiagonal elements of the unit bidiagonal matrix *L* are contained in elements 1 to *n*-1 of *L* if the matrix is not splitted. At the end of each block the corresponding shift is stored as given by slarre for real flavors and by dlarre for double precision flavors.

pivmin REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 The minimum pivot allowed in the Sturm sequence.

isplit INTEGER. Array, DIMENSION (*n*).
 The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), etc.

m INTEGER. The total number of eigenvalues found.
 $0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I', $m = i_u - i_l + 1$.

dol, dou INTEGER.
 If you want to compute only selected eigenvectors from all the eigenvalues supplied, specify an index range *dol:dou*. Or else apply the setting *dol*=1, *dou*=*m*. Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.
 If you want to compute only selected eigenpairs, then the columns *dol*-1 to *dou*+1 of the eigenvector space *Z* contain the computed eigenvectors. All other columns of *Z* are set to zero.

minrgp, rtol1, rtol2 REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv
 Parameters for bisection. An interval [LEFT,RIGHT] has converged if $\text{RIGHT} - \text{LEFT} < \text{MAX}(\text{rtol1} * \text{gap}, \text{rtol2} * \max(|\text{LEFT}|, |\text{RIGHT}|))$.

w REAL for slarrv/clarrv
 DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (n). The first m elements of w contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (the output array w from ?larre is expected here). These eigenvalues are set with respect to the shift of the corresponding root representation for their block.

werr REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (n). The first m elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in w .

wgap REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION (n). The separation from the right neighbor eigenvalue in w .

iblock INTEGER. Array, DIMENSION (n).

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $=2$ if $w(i)$ belongs to the second block, etc.

indexw INTEGER. Array, DIMENSION (n).

The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION ($2*n$). The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$). The Gerschgorin intervals should be computed from the original unshifted matrix.

ldz INTEGER. The leading dimension of the output array Z . $ldz \geq 1$, and if $jobz = 'V'$, $ldz \geq \max(1, n)$.

work REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Workspace array, DIMENSION ($12*n$).

iwork INTEGER.

Workspace array, DIMENSION ($7*n$).

Output Parameters

d On exit, d may be overwritten.

l On exit, l is overwritten.

<i>w</i>	On exit, <i>w</i> holds the eigenvalues of the unshifted matrix.
<i>werr</i>	On exit, <i>werr</i> contains refined values of its input approximations.
<i>wgap</i>	On exit, <i>wgap</i> contains refined values of its input approximations. Very small gaps are changed.
<i>z</i>	<p>REAL for <i>slarrv</i></p> <p>DOUBLE PRECISION for <i>dlarrv</i></p> <p>COMPLEX for <i>clarrv</i></p> <p>DOUBLE COMPLEX for <i>zlarrv</i></p> <p>Array, DIMENSION (<i>ldz</i>, max(1,<i>m</i>)).</p> <p>If <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the input eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>).</p>

NOTE

The user must ensure that at least max(1,*m*) columns are supplied in the array *z*.

<i>isuppz</i>	<p>INTEGER .</p> <p>Array, DIMENSION (2*max(1,<i>m</i>)). The support of the eigenvectors in <i>z</i>, that is, the indices indicating the nonzero elements in <i>z</i>. The <i>i</i>-th eigenvector is nonzero only in elements <i>isuppz</i>(2<i>i</i>-1) through <i>isuppz</i>(2<i>i</i>).</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0: successful exit</p> <p>If <i>info</i> > 0: A problem occurred in ?<i>larrv</i>. If <i>info</i> = 5, the Rayleigh Quotient Iteration failed to converge to full accuracy.</p> <p>If <i>info</i> < 0: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter <i>info</i> for further information is required.</p> <ul style="list-style-type: none"> • If <i>info</i> = -1, there is a problem in ?<i>larrb</i> when refining a child eigenvalue; • If <i>info</i> = -2, there is a problem in ?<i>larrf</i> when computing the relatively robust representation (RRR) of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter <i>minrgp</i> smaller and recompile. However, as the orthogonality of the computed vectors is proportional to 1/<i>minrgp</i>, you should be aware that you might be trading in precision when you decrease <i>minrgp</i>. • If <i>info</i> = -3, there is a problem in ?<i>larrb</i> when refining a single eigenvalue after the Rayleigh correction was rejected.

See Also

?*larrb*
?*larre*
?*larrf*

?lartg

Generates a plane rotation with real cosine and real/complex sine.

Syntax

```
call slartg( f, g, cs, sn, r )
call dlartg( f, g, cs, sn, r )
call clartg( f, g, cs, sn, r )
call zlartg( f, g, cs, sn, r )
```

Include Files

- mkl.fi

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -\text{conjg}(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine [?rotg](#), except for the following differences.

For [slartg](#)/[dlartg](#):

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=1$ without doing any floating point operations (saves work in [?bdsqr](#) when there are zeros on the diagonal);

If f exceeds g in magnitude, cs will be positive.

For [clartg](#)/[zlartg](#):

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$, then $cs=0$ and sn is chosen so that r is real.

Input Parameters

f, g	REAL for slartg
	DOUBLE PRECISION for dlartg
	COMPLEX for clartg
	DOUBLE COMPLEX for zlartg
	The first and second component of vector to be rotated.

Output Parameters

<i>cs</i>	REAL for slartg/clartg DOUBLE PRECISION for dlartg/zlartg The cosine of the rotation.
<i>sn</i>	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg The sine of the rotation.
<i>r</i>	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg The nonzero component of the rotated vector.

?lartgp

Generates a plane rotation.

Syntax

call slartgp(*f*, *g*, *cs*, *sn*, *r*)

call dlartgp(*f*, *g*, *cs*, *sn*, *r*)

call lartgp(*f*, *g*, *cs*, *sn*, *r*)

Include Files

- mkl.fi

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + sn^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine ?rotg, except for the following differences:

- *f* and *g* are unchanged on return.
- If *g*=0, then *cs*=(+/-)1 and *sn*=0.
- If *f*=0 and *g*≠ 0, then *cs*=0 and *sn*=(+/-)1.

The sign is chosen so that $r \geq 0$.

Input Parameters

The data types are given for the Fortran interface.

f, *g* REAL for slartgp
 DOUBLE PRECISION for dlartgp
 The first and second component of the vector to be rotated.

Output Parameters

cs REAL for slartgp
 DOUBLE PRECISION for dlartgp
 The cosine of the rotation.

sn REAL for slartgp
 DOUBLE PRECISION for dlartgp
 The sine of the rotation.

r REAL for slartgp
 DOUBLE PRECISION for dlartgp
 The nonzero component of the rotated vector.

info INTEGER. If *info* = 0, the execution is successful.
 If *info* = -1, *f* is NaN.
 If *info* = -2, *g* is NaN.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine ?lartgp interface are as follows:

f Holds the first component of the vector to be rotated.

g Holds the second component of the vector to be rotated.

cs Holds the cosine of the rotation.

sn Holds the sine of the rotation.

r Holds the nonzero component of the rotated vector.

See Also

?rotg
 ?lartg
 ?lartgs

zlartgs

Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Syntax

```
call slartgs( x, y, sigma, cs, sn )
call dlartgs( x, y, sigma, cs, sn )
call lartgs( x, y, sigma, cs, sn )
```

Include Files

- mkl.fi

Description

The routine generates a plane rotation designed to introduce a bulge in Golub-Reinsch-style implicit QR iteration for the bidiagonal SVD problem. x and y are the top-row entries, and $sigma$ is the shift. The computed cs and sn define a plane rotation that satisfies the following:

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} x^2 - sigma \\ x * y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

with r nonnegative.

If $x^2 - sigma$ and $x * y$ are 0, the rotation is by $\pi/2$

Input Parameters

The data types are given for the Fortran interface.

x, y	REAL for slartgs DOUBLE PRECISION for dlartgs The (1,1) and (1,2) entries of an upper bidiagonal matrix, respectively.
$sigma$	REAL for slartgs DOUBLE PRECISION for dlartgs Shift

Output Parameters

cs	REAL for slartgs DOUBLE PRECISION for dlartgs The cosine of the rotation.
sn	REAL for slartgs DOUBLE PRECISION for dlartgs The sine of the rotation.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `?lartgs` interface are as follows:

<code>x</code>	Holds the (1,1) entry of an upper diagonal matrix.
<code>y</code>	Holds the (1,2) entry of an upper diagonal matrix.
<code>sigma</code>	Holds the shift.
<code>cs</code>	Holds the cosine of the rotation.
<code>sn</code>	Holds the sine of the rotation.

See Also

[?lartg](#)

[?lartgp](#)

?lartv

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slartv( n, x, incx, y, incy, c, s, incc )
call dlartv( n, x, incx, y, incy, c, s, incc )
call clartv( n, x, incx, y, incy, c, s, incc )
call zlartv( n, x, incx, y, incy, c, s, incc )
```

Include Files

- `mkl.fi`

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors `x` and `y`. For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\text{conjg}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

<code>n</code>	INTEGER. The number of plane rotations to be applied.
<code>x, y</code>	REAL for <code>slartv</code> DOUBLE PRECISION for <code>dlartv</code>

	COMPLEX for clartv
	DOUBLE COMPLEX for zartv
	Arrays, DIMENSION $(1+(n-1)*incx)$ and $(1+(n-1)*incy)$, respectively. The input vectors x and y .
<i>incx</i>	INTEGER. The increment between elements of x . $incx > 0$.
<i>incy</i>	INTEGER. The increment between elements of y . $incy > 0$.
<i>c</i>	REAL for slartv/clartv DOUBLE PRECISION for dlartv/zlartv Array, DIMENSION $(1+(n-1)*incc)$. The cosines of the plane rotations.
<i>s</i>	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv DOUBLE COMPLEX for zartv Array, DIMENSION $(1+(n-1)*incc)$. The sines of the plane rotations.
<i>incc</i>	INTEGER. The increment between elements of c and s . $incc > 0$.

Output Parameters

x, y The rotated vectors x and y .

?laruv

Returns a vector of n random real numbers from a uniform distribution.

Syntax

```
call slaruv( iseed, n, x )
```

```
call dlaruv( iseed, n, x )
```

Include Files

- mkl.fi

Description

The routine ?laruv returns a vector of n random real numbers from a uniform (0,1) distribution ($n \leq 128$).

This is an auxiliary routine called by ?larnv.

Input Parameters

<i>iseed</i>	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <i>iseed</i> (4) must be odd.
<i>n</i>	INTEGER. The number of random numbers to be generated. $n \leq 128$.

Output Parameters

<i>x</i>	REAL for slaruv DOUBLE PRECISION for dlaruv Array, DIMENSION (<i>n</i>). The generated random numbers.
<i>seed</i>	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by ?tzzrf) to a general matrix.

Syntax

```
call slarz( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz( side, m, n, l, v, incv, tau, c, ldc, work )
```

Include Files

- mkl.fi

Description

The routine ?larz applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in the forms

$H = I - \tau v v^T$ for real flavors and $H = I - \tau v v^H$ for complex flavors,

where τ is a real/complex scalar and v is a real/complex vector, respectively.

If $\tau = 0$, then H is taken to be the unit matrix.

For complex flavors, to apply H^H (the conjugate transpose of H), supply $\text{conj}(\tau)$ instead of τ .

H is a product of k elementary reflectors as returned by ?tzzrf.

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form H^*C If <i>side</i> = 'R': form C^*H
<i>m</i>	INTEGER. The number of rows of the matrix C .

<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> .
<i>l</i>	<p>INTEGER. The number of entries of the vector <i>v</i> containing the meaningful part of the Householder vectors.</p> <p>If <i>side</i> = 'L', $m \geq l \geq 0$,</p> <p>if <i>side</i> = 'R', $n \geq l \geq 0$.</p>
<i>v</i>	<p>REAL for slarz</p> <p>DOUBLE PRECISION for dlarz</p> <p>COMPLEX for clarz</p> <p>DOUBLE COMPLEX for zlarz</p> <p>Array, DIMENSION (1+(<i>l</i>-1)*abs(<i>incv</i>)).</p> <p>The vector <i>v</i> in the representation of <i>H</i> as returned by ?tzrzf.</p> <p><i>v</i> is not used if <i>tau</i> = 0.</p>
<i>incv</i>	<p>INTEGER. The increment between elements of <i>v</i>.</p> <p><i>incv</i> ≠ 0.</p>
<i>tau</i>	<p>REAL for slarz</p> <p>DOUBLE PRECISION for dlarz</p> <p>COMPLEX for clarz</p> <p>DOUBLE COMPLEX for zlarz</p> <p>The value <i>tau</i> in the representation of <i>H</i>.</p>
<i>c</i>	<p>REAL for slarz</p> <p>DOUBLE PRECISION for dlarz</p> <p>COMPLEX for clarz</p> <p>DOUBLE COMPLEX for zlarz</p> <p>Array, DIMENSION (<i>ldc</i>,<i>n</i>).</p> <p>On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.</p>
<i>ldc</i>	<p>INTEGER. The leading dimension of the array <i>c</i>.</p> <p><i>ldc</i> ≥ max(1, <i>m</i>).</p>
<i>work</i>	<p>REAL for slarz</p> <p>DOUBLE PRECISION for dlarz</p> <p>COMPLEX for clarz</p> <p>DOUBLE COMPLEX for zlarz</p> <p>Workspace array, DIMENSION</p> <p>(<i>n</i>) if <i>side</i> = 'L' or</p> <p>(<i>m</i>) if <i>side</i> = 'R'.</p>

Output Parameters

c On exit, *C* is overwritten by the matrix H^*C if *side* = 'L', or C^*H if *side* = 'R'.

?larzb

Applies a block reflector or its transpose/conjugate-transpose to a general matrix.

Syntax

```
call slarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

```
call dlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

```
call clarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

```
call zlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

Include Files

- mkl.fi

Description

The routine applies a real/complex block reflector H or its transpose H^T (or the conjugate transpose H^H for complex flavors) to a real/complex distributed m -by- n matrix C from the left or the right. Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

side CHARACTER*1.
 If *side* = 'L': apply H or H^T/H^H from the left
 If *side* = 'R': apply H or H^T/H^H from the right

trans CHARACTER*1.
 If *trans* = 'N': apply H (No transpose)
 If *trans*='C': apply H^H (conjugate transpose)
 If *trans*='T': apply H^T (transpose transpose)

direct CHARACTER*1.
 Indicates how H is formed from a product of elementary reflectors
 = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward, not supported)
 = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)

storev CHARACTER*1.
 Indicates how the vectors which define the elementary reflectors are stored:

	<p>= 'C': Column-wise (not supported)</p> <p>= 'R': Row-wise.</p>
<i>m</i>	INTEGER. The number of rows of the matrix <i>C</i> .
<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. The order of the matrix <i>T</i> (equal to the number of elementary reflectors whose product defines the block reflector).
<i>l</i>	<p>INTEGER. The number of columns of the matrix <i>V</i> containing the meaningful part of the Householder reflectors.</p> <p>If <i>side</i> = 'L', $m \geq l \geq 0$, if <i>side</i> = 'R', $n \geq l \geq 0$.</p>
<i>v</i>	<p>REAL for slarzb</p> <p>DOUBLE PRECISION for dlarzb</p> <p>COMPLEX for clarzb</p> <p>DOUBLE COMPLEX for zlarzb</p> <p>Array, DIMENSION (<i>ldv</i>, <i>nv</i>).</p> <p>If <i>storev</i> = 'C', <i>nv</i> = <i>k</i>;</p> <p>if <i>storev</i> = 'R', <i>nv</i> = <i>l</i>.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i>.</p> <p>If <i>storev</i> = 'C', $ldv \geq l$; if <i>storev</i> = 'R', $ldv \geq k$.</p>
<i>t</i>	<p>REAL for slarzb</p> <p>DOUBLE PRECISION for dlarzb</p> <p>COMPLEX for clarzb</p> <p>DOUBLE COMPLEX for zlarzb</p> <p>Array, DIMENSION (<i>ldt</i>, <i>k</i>). The triangular <i>k</i>-by-<i>k</i> matrix <i>T</i> in the representation of the block reflector.</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of the array <i>t</i>.</p> <p>$ldt \geq k$.</p>
<i>c</i>	<p>REAL for slarzb</p> <p>DOUBLE PRECISION for dlarzb</p> <p>COMPLEX for clarzb</p> <p>DOUBLE COMPLEX for zlarzb</p> <p>Array, DIMENSION (<i>ldc</i>, <i>n</i>). On entry, the <i>m</i>-by-<i>n</i> matrix <i>C</i>.</p>
<i>ldc</i>	<p>INTEGER. The leading dimension of the array <i>c</i>.</p> <p>$ldc \geq \max(1, m)$.</p>
<i>work</i>	REAL for slarzb

DOUBLE PRECISION for dlarzb

COMPLEX for clarzb

DOUBLE COMPLEX for zlarzb

Workspace array, DIMENSION (*ldwork*, *k*).

ldwork

INTEGER. The leading dimension of the array *work*.

If *side* = 'L', $ldwork \geq \max(1, n)$;

if *side* = 'R', $ldwork \geq \max(1, m)$.

Output Parameters

c

On exit, *C* is overwritten by H^*C , or H^T/H^H*C , or C^*H , or C^*H^T/H^H .

?larzt

Forms the triangular factor *T* of a block reflector $H = I - V^*T^*V^H$.

Syntax

call slarzt(*direct*, *storev*, *n*, *k*, *v*, *ldv*, *tau*, *t*, *ldt*)

call dlarzt(*direct*, *storev*, *n*, *k*, *v*, *ldv*, *tau*, *t*, *ldt*)

call clarzt(*direct*, *storev*, *n*, *k*, *v*, *ldv*, *tau*, *t*, *ldt*)

call zlarzt(*direct*, *storev*, *n*, *k*, *v*, *ldv*, *tau*, *t*, *ldt*)

Include Files

- mkl.fi

Description

The routine forms the triangular factor *T* of a real/complex block reflector *H* of order $> n$, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1)*H(2)*\dots*H(k)$, and *T* is upper triangular.

If *direct* = 'B', $H = H(k)*\dots*H(2)*H(1)$, and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the array *v*, and $H = I - V^*T^*V^T$ (for real flavors) or $H = I - V^*T^*V^H$ (for complex flavors).

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th row of the array *v*, and $H = I - V^T*T^*V$ (for real flavors) or $H = I - V^H*T^*V$ (for complex flavors).

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct

CHARACTER*1.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

If *direct* = 'F': $H = H(1)*H(2)*\dots*H(k)$ (forward, not supported)

	If <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below): If <i>storev</i> = 'C': column-wise (not supported) If <i>storev</i> = 'R': row-wise
<i>n</i>	INTEGER. The order of the block reflector <i>H</i> . $n \geq 0$.
<i>k</i>	INTEGER. The order of the triangular factor <i>T</i> (equal to the number of elementary reflectors). $k \geq 1$.
<i>v</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt DOUBLE COMPLEX for zlarzt Array, DIMENSION (<i>ldv</i> , <i>k</i>) if <i>storev</i> = 'C' (<i>ldv</i> , <i>n</i>) if <i>storev</i> = 'R' The matrix <i>V</i> .
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> . If <i>storev</i> = 'C', $ldv \geq \max(1, n)$; if <i>storev</i> = 'R', $ldv \geq k$.
<i>tau</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt DOUBLE COMPLEX for zlarzt Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector <i>H</i> (<i>i</i>).
<i>ldt</i>	INTEGER. The leading dimension of the output array <i>t</i> . $ldt \geq k$.

Output Parameters

<i>t</i>	REAL for slarzt DOUBLE PRECISION for dlarzt COMPLEX for clarzt DOUBLE COMPLEX for zlarzt Array, DIMENSION (<i>ldt</i> , <i>k</i>). The <i>k</i> -by- <i>k</i> triangular factor <i>T</i> of the block reflector. If <i>direct</i> = 'F', <i>T</i> is upper triangular; if <i>direct</i> = 'B', <i>T</i> is lower triangular. The rest of the array is not used.
----------	---

v The matrix V . See *Application Notes* below.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

`direct = 'F' and storev = 'C':` `direct = 'F' and storev = 'R':`

$$v = \begin{bmatrix} v_1 & v_2 & v_3 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{bmatrix}$$

$$\begin{array}{c} \text{---}V\text{---} \\ / \quad \quad \backslash \\ \left[\begin{array}{ccccccccc} v_1 & v_1 & v_1 & v_1 & v_1 & \cdot & \cdot & \cdot & \cdot & 1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & \cdot & \cdot & \cdot & \cdot & 1 \\ v_3 & v_3 & v_3 & v_3 & v_3 & \cdot & \cdot & \cdot & 1 & \cdot \end{array} \right] \end{array}$$

`direct = 'B' and storev = 'C':` `direct = 'B' and storev = 'R':`

$$v = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

$$\begin{array}{c} \text{---}V\text{---} \\ / \quad \quad \backslash \\ \left[\begin{array}{ccccccccc} 1 & \cdot & \cdot & \cdot & \cdot & v_1 & v_1 & v_1 & v_1 & v_1 \\ \cdot & 1 & \cdot & \cdot & \cdot & v_2 & v_2 & v_2 & v_2 & v_2 \\ \cdot & \cdot & 1 & \cdot & \cdot & v_3 & v_3 & v_3 & v_3 & v_3 \end{array} \right] \end{array}$$

?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

```
call slas2( f, g, h, ssmín, ssmáx )  
call dlas2( f, g, h, ssmín, ssmáx )
```

Include Files

- mkl.fi

Description

The routine ?las2 computes the singular values of the 2-by-2 matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, *ssmin* is the smaller singular value and *SSMAX* is the larger singular value.

Input Parameters

<i>f, g, h</i>	REAL for slas2
	DOUBLE PRECISION for dlas2
	The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

<i>ssmin, ssmáx</i>	REAL for slas2
	DOUBLE PRECISION for dlas2
	The smaller and the larger singular values, respectively.

Application Notes

Barring over/underflow, all output quantities are correct to within a few units in the last place (*ulps*), even in the absence of a guard digit in addition/subtraction. In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .

Syntax

```
call slascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

```
call dlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

```
call clascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

```
call zlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine `?lascl` multiplies the m -by- n real/complex matrix A by the real scalar c_{to}/c_{from} . The operation is performed without over/underflow as long as the final result $c_{to} * A(i, j) / c_{from}$ does not over/underflow.

`type` specifies that A may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

<code>type</code>	CHARACTER*1. This parameter specifies the storage type of the input matrix. = 'G': A is a full matrix. = 'L': A is a lower triangular matrix. = 'U': A is an upper triangular matrix. = 'H': A is an upper Hessenberg matrix. = 'B': A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the lower half stored = 'Q': A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the upper half stored. = 'Z': A is a band matrix with lower bandwidth kl and upper bandwidth ku . See description of the <code>?gbtrf</code> function for storage details.
<code>kl</code>	INTEGER. The lower bandwidth of A . Referenced only if <code>type</code> = 'B', 'Q' or 'Z'.
<code>ku</code>	INTEGER. The upper bandwidth of A . Referenced only if <code>type</code> = 'B', 'Q' or 'Z'.
<code>cfrom, cto</code>	REAL for <code>slascl/clascl</code> DOUBLE PRECISION for <code>dlascl/zlascl</code> The matrix A is multiplied by $cto/cfrom$. $A(i, j)$ is computed without over/underflow if the final result $cto * A(i, j) / cfrom$ can be represented without over/underflow. <code>cfrom</code> must be nonzero.
<code>m</code>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<code>n</code>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<code>a</code>	REAL for <code>slascl</code> DOUBLE PRECISION for <code>dlascl</code>

COMPLEX for `clascl`

DOUBLE COMPLEX for `zlascl`

Array, DIMENSION (*lda*, *n*). The matrix to be multiplied by *cto/cfrom*. See *type* for the storage type.

lda INTEGER. The leading dimension of the array *a*.

$lda \geq \max(1, m)$.

Output Parameters

a The multiplied matrix *A*.

info INTEGER.

If *info* = 0 - successful exit

If *info* = -*i* < 0, the *i*-th argument had an illegal value.

See Also

[?gbtrf](#)

?lasd0

Computes the singular values of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e . Used by [?bdsdc](#).

Syntax

```
call slasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
```

```
call dlasd0( n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info )
```

Include Files

- `mkl.fi`

Description

Using a divide and conquer approach, the routine `?lasd0` computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and offdiagonal e , where $m = n + sqre$.

The algorithm computes orthogonal matrices U and VT such that $B = U^*S^*VT$. The singular values S are overwritten on d .

The related subroutine `?lasda` computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .

sqre INTEGER. Specifies the column dimension of the bidiagonal matrix.

If *sqre* = 0: the bidiagonal matrix has column dimension $m = n$.

If *sqre* = 1: the bidiagonal matrix has column dimension $m = n+1$.

<i>d</i>	REAL for <code>slasd0</code> DOUBLE PRECISION for <code>dlsd0</code> Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for <code>slasd0</code> DOUBLE PRECISION for <code>dlsd0</code> Array, DIMENSION (<i>m</i> -1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>ldu</i>	INTEGER. On entry, leading dimension of the output array <i>u</i> .
<i>ldvt</i>	INTEGER. On entry, leading dimension of the output array <i>vt</i> .
<i>smlsiz</i>	INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.
<i>iwork</i>	INTEGER. Workspace array, dimension must be at least $(8n)$.
<i>work</i>	REAL for <code>slasd0</code> DOUBLE PRECISION for <code>dlsd0</code> Workspace array, dimension must be at least $(3m^2+2m)$.

Output Parameters

<i>d</i>	On exit <i>d</i> , If <i>info</i> = 0, contains singular values of the bidiagonal matrix.
<i>u</i>	REAL for <code>slasd0</code> DOUBLE PRECISION for <code>dlsd0</code> Array, DIMENSION at least (<i>ldu</i> , <i>n</i>). On exit, <i>u</i> contains the left singular vectors.
<i>vt</i>	REAL for <code>slasd0</code> DOUBLE PRECISION for <code>dlsd0</code> Array, DIMENSION at least (<i>ldvt</i> , <i>m</i>). On exit, <i>vt</i> ^T contains the right singular vectors.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value. If <i>info</i> = 1, a singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix *B* of the specified size. Used by ?bdsdc.

Syntax

```
call slasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
call dlasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
```

Include Files

- mkl.fi

Description

The routine computes the SVD of an upper bidiagonal n -by- m matrix B , where $n = nl + nr + 1$ and $m = n + sqre$.

The routine `?lasd1` is called from `?lasd0`.

A related subroutine `?lasd7` handles the case in which the singular values (and the singular vectors in factored form) are desired.

`?lasd1` computes the SVD as follows:

$$VT = U(in) * \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1^T & a & Z2^T & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$= U(out) * (D(out) \ 0) * VT(out)$

where $Z^T = (Z1^T a Z2^T b) = u^T * VT^T$, and u is a vector of dimension m with $alpha$ and $beta$ in the $nl+1$ and $nl+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The left singular vectors of the original matrix are stored in u , and the transpose of the right singular vectors are stored in vt , and the singular values are in d . The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the Z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd2`.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine `?lasd4` (as called by `?lasd3`). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl INTEGER. The row dimension of the upper block.
 $nl \geq 1$.

nr INTEGER. The row dimension of the lower block.
 $nr \geq 1$.

sqre INTEGER.
 If $sqre = 0$: the lower block is an nr -by- nr square matrix.

If $s_{qre} = 1$: the lower block is an nr -by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + s_{qre}$.

<i>d</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Array, DIMENSION $(nl+nr+1)$. $n = nl+nr+1$. On entry $d(1:nl, 1:nl)$ contains the singular values of the upper block; and $d(nl+2:n)$ contains the singular values of the lower block.
<i>alpha</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Contains the off-diagonal element associated with the added row.
<i>u</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Array, DIMENSION (ldu, n) . On entry $u(1:nl, 1:nl)$ contains the left singular vectors of the upper block; $u(nl+2:n, nl+2:n)$ contains the left singular vectors of the lower block.
<i>ldu</i>	INTEGER. The leading dimension of the array U . $ldu \geq \max(1, n)$.
<i>vt</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Array, DIMENSION $(ldvt, m)$, where $m = n + s_{qre}$. On entry $vt(1:nl+1, 1:nl+1)^T$ contains the right singular vectors of the upper block; $vt(nl+2:m, nl+2:m)^T$ contains the right singular vectors of the lower block.
<i>ldvt</i>	INTEGER. The leading dimension of the array vt . $ldvt \geq \max(1, M)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION $(4n)$.
<i>work</i>	REAL for slasd1 DOUBLE PRECISION for dlasd1 Workspace array, DIMENSION $(3m_2 + 2m)$.

Output Parameters

d On exit $d(1:n)$ contains the singular values of the modified matrix.

<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>u</i>	On exit <i>u</i> contains the left singular vectors of the bidiagonal matrix.
<i>vt</i>	On exit <i>vt</i> ^T contains the right singular vectors of the bidiagonal matrix.
<i>idxq</i>	INTEGER Array, DIMENSION (<i>n</i>). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, <i>d</i> (<i>idxq</i> (<i>i</i> = 1, <i>n</i>)) will be in ascending order.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value. If <i>info</i> = 1, a singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

```
call slasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma, u2, ldu2,
vt2, ldvt2, idxp, idx, idxp, idxq, coltyp, info )
```

```
call dlasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma, u2, ldu2,
vt2, ldvt2, idxp, idx, idxp, idxq, coltyp, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine ?lasd2 is called from ?lasd1.

Input Parameters

<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.

sqre INTEGER.
 If *sqre* = 0): the lower block is an *nr*-by-*nr* square matrix
 If *sqre* = 1): the lower block is an *nr*-by-(*nr*+1) rectangular matrix. The bidiagonal matrix has $n = n1 + nr + 1$ rows and $m = n + sqre \geq n$ columns.

d REAL for *slasd2*
 DOUBLE PRECISION for *dlsd2*
 Array, DIMENSION (*n*). On entry *d* contains the singular values of the two submatrices to be combined.

alpha REAL for *slasd2*
 DOUBLE PRECISION for *dlsd2*
 Contains the diagonal element associated with the added row.

beta REAL for *slasd2*
 DOUBLE PRECISION for *dlsd2*
 Contains the off-diagonal element associated with the added row.

u REAL for *slasd2*
 DOUBLE PRECISION for *dlsd2*
 Array, DIMENSION (*ldu*, *n*). On entry *u* contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*, *nl*), and (*nl*+2, *nl*+2), (*n*,*n*).

ldu INTEGER. The leading dimension of the array *u*.
ldu $\geq n$.

ldu2 INTEGER. The leading dimension of the output array *u2*. *ldu2* $\geq n$.

vt REAL for *slasd2*
 DOUBLE PRECISION for *dlsd2*
 Array, DIMENSION (*ldvt*, *m*). On entry, *vt*^T contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (*nl*+1, *nl*+1), and (*nl*+2, *nl*+2), (*m*, *m*).

ldvt INTEGER. The leading dimension of the array *vt*. *ldvt* $\geq m$.

ldvt2 INTEGER. The leading dimension of the output array *vt2*. *ldvt2* $\geq m$.

idxp INTEGER.
 Workspace array, DIMENSION (*n*). This will contain the permutation used to place deflated values of *D* at the end of the array. On output *idxp*(2:*k*) points to the nondeflated *d*-values and *idxp*(*k*+1:*n*) points to the deflated singular values.

idx INTEGER.

Workspace array, `DIMENSION (n)`. This will contain the permutation used to sort the contents of d into ascending order.

coltyp

INTEGER.

Workspace array, `DIMENSION (n)`. As workspace, this array contains a label that indicates which of the following types a column in the $u2$ matrix or a row in the $vt2$ matrix is:

1 : non-zero in the upper half only

2 : non-zero in the lower half only

3 : dense

4 : deflated.

idxq

INTEGER. Array, `DIMENSION (n)`. This parameter contains the permutation that separately sorts the two sub-problems in D in the ascending order. Note that entries in the first half of this permutation must first be moved one position backwards and entries in the second half must have $n1+1$ added to their values.

Output Parameters

k

INTEGER. Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \leq k \leq n$.

d

On exit D contains the trailing $(n-k)$ updated singular values (those which were deflated) sorted into increasing order.

u

On exit u contains the trailing $(n-k)$ updated left singular vectors (those which were deflated) in its last $n-k$ columns.

z

REAL for `s1asd2`

DOUBLE PRECISION for `d1asd2`

Array, `DIMENSION (n)`. On exit, z contains the updating row vector in the secular equation.

dsigma

REAL for `s1asd2`

DOUBLE PRECISION for `d1asd2`

Array, `DIMENSION (n)`. Contains a copy of the diagonal elements ($k-1$ singular values and one zero) in the secular equation.

u2

REAL for `s1asd2`

DOUBLE PRECISION for `d1asd2`

Array, `DIMENSION (Idu2, n)`. Contains a copy of the first $k-1$ left singular vectors which will be used by `?1asd3` in a matrix multiply (`?gemm`) to solve for the new left singular vectors. $u2$ is arranged into four blocks. The first block contains a column with 1 at $n1+1$ and zero everywhere else; the second block contains non-zero entries only at and above $n1$; the third contains non-zero entries only below $n1+1$; and the fourth is dense.

<code>vt</code>	On exit, vt^T contains the trailing $(n-k)$ updated right singular vectors (those which were deflated) in its last $n-k$ columns. In case $sqre = 1$, the last row of vt spans the right null space.
<code>vt2</code>	REAL for <code>slasd2</code> DOUBLE PRECISION for <code>dlsasd2</code> Array, DIMENSION $(ldvt2, n)$. $vt2^T$ contains a copy of the first k right singular vectors which will be used by <code>?lasd3</code> in a matrix multiply (<code>?gemm</code>) to solve for the new right singular vectors. $vt2$ is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in σ ; the second block contains non-zeros only at and before $nl + 1$; the third block contains non-zeros only at and after $nl + 2$.
<code>idxc</code>	INTEGER. Array, DIMENSION (n) . This will contain the permutation used to arrange the columns of the deflated u matrix into three groups: the first group contains non-zero entries only at and above nl , the second contains non-zero entries only below $nl+2$, and the third is dense.
<code>coltyp</code>	On exit, it is an array of dimension 4, with $coltyp(i)$ being the dimension of the i -th type columns.
<code>info</code>	INTEGER. If $info = 0$): successful exit If $info = -i < 0$, the i -th argument had an illegal value.

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in D and Z , and then updates the singular vectors by matrix multiplication. Used by `?bdsdc`.

Syntax

```
call slasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2,
            ldvt2, idxc, ctot, z, info )
```

```
call dlsasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2,
             ldvt2, idxc, ctot, z, info )
```

Include Files

- `mk1.fi`

Description

The routine `?lasd3` finds all the square roots of the roots of the secular equation, as defined by the values in D and Z .

It makes the appropriate calls to `?lasd4` and then updates the singular vectors by matrix multiplication.

The routine `?lasd3` is called from `?lasd1`.

Input Parameters

<i>nl</i>	<p>INTEGER. The row dimension of the upper block.</p> <p>$nl \geq 1$.</p>
<i>nr</i>	<p>INTEGER. The row dimension of the lower block.</p> <p>$nr \geq 1$.</p>
<i>sqre</i>	<p>INTEGER.</p> <p>If $sqre = 0$): the lower block is an nr-by-nr square matrix.</p> <p>If $sqre = 1$): the lower block is an nr-by-$(nr+1)$ rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.</p>
<i>k</i>	<p>INTEGER. The size of the secular equation, $1 \leq k \leq n$.</p>
<i>q</i>	<p>REAL for <code>slasd3</code></p> <p>DOUBLE PRECISION for <code>dlsasd3</code></p> <p>Workspace array, DIMENSION at least (<i>ldq</i>, <i>k</i>).</p>
<i>ldq</i>	<p>INTEGER. The leading dimension of the array <i>Q</i>.</p> <p>$ldq \geq k$.</p>
<i>dsigma</i>	<p>REAL for <code>slasd3</code></p> <p>DOUBLE PRECISION for <code>dlsasd3</code></p> <p>Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.</p>
<i>ldu</i>	<p>INTEGER. The leading dimension of the array <i>u</i>.</p> <p>$ldu \geq n$.</p>
<i>u2</i>	<p>REAL for <code>slasd3</code></p> <p>DOUBLE PRECISION for <code>dlsasd3</code></p> <p>Array, DIMENSION (<i>ldu2</i>, <i>n</i>).</p> <p>The first <i>k</i> columns of this matrix contain the non-deflated left singular vectors for the split problem.</p>
<i>ldu2</i>	<p>INTEGER. The leading dimension of the array <i>u2</i>.</p> <p>$ldu2 \geq n$.</p>
<i>ldvt</i>	<p>INTEGER. The leading dimension of the array <i>vt</i>.</p> <p>$ldvt \geq n$.</p>
<i>vt2</i>	<p>REAL for <code>slasd3</code></p> <p>DOUBLE PRECISION for <code>dlsasd3</code></p> <p>Array, DIMENSION (<i>ldvt2</i>, <i>n</i>).</p>

The first k columns of $vt2'$ contain the non-deflated right singular vectors for the split problem.

ldvt2

INTEGER. The leading dimension of the array *vt2*.

$ldvt2 \geq n$.

idxc

INTEGER. Array, DIMENSION (n).

The permutation used to arrange the columns of u (and rows of vt) into three groups: the first group contains non-zero entries only at and above (or before) $n1 + 1$; the second contains non-zero entries only at and below (or after) $n1+2$; and the third is dense. The first column of u and the row of vt are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.

ctot

INTEGER. Array, DIMENSION (4). A count of the total number of the various types of columns in u (or rows in vt), as described in *idxc*.

The fourth column type is any column which has been deflated.

z

REAL for *slasd3*

DOUBLE PRECISION for *dlasd3*

Array, DIMENSION (k). The first k elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d

REAL for *slasd3*

DOUBLE PRECISION for *dlasd3*

Array, DIMENSION (k). On exit the square roots of the roots of the secular equation, in ascending order.

u

REAL for *slasd3*

DOUBLE PRECISION for *dlasd3*

Array, DIMENSION (ldu, n).

The last $n - k$ columns of this matrix contain the deflated left singular vectors.

vt

REAL for *slasd3*

DOUBLE PRECISION for *dlasd3*

Array, DIMENSION ($ldvt, m$).

The last $m - k$ columns of vt' contain the deflated right singular vectors.

vt2

Destroyed on exit.

z

Destroyed on exit.

info

INTEGER.

If *info* = 0): successful exit.

If $info = -i < 0$, the i -th argument had an illegal value.

If $info = 1$, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.

Syntax

```
call slasd4( n, i, d, z, delta, rho, sigma, work, info)
call dlasd4( n, i, d, z, delta, rho, sigma, work, info )
```

Include Files

- mkl.fi

Description

The routine computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array d , and that $0 \leq d(i) < d(j)$ for $i < j$ and that $rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(d) * \text{diag}(d) + rho * Z * Z^T,$$

where the Euclidean norm of Z is equal to 1. The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	INTEGER. The length of all arrays.
i	INTEGER. The index of the eigenvalue to be computed. $1 \leq i \leq n$.
d	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (n). The original eigenvalues. They must be in order, $0 \leq d(i) < d(j)$ for $i < j$.
z	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (n).

The components of the updating vector.

rho REAL for slasd4
DOUBLE PRECISION for dlasd4
The scalar in the symmetric updating formula.

work REAL for slasd4
DOUBLE PRECISION for dlasd4
Workspace array, DIMENSION (*n*).
If $n \neq 1$, *work* contains $(d(j) + \text{sigma}_i)$ in its *j*-th component.
If $n = 1$, then $\text{work}(1) = 1$.

Output Parameters

delta REAL for slasd4
DOUBLE PRECISION for dlasd4
Array, DIMENSION (*n*).
If $n \neq 1$, *delta* contains $(d(j) - \text{sigma}_i)$ in its *j*-th component.
If $n = 1$, then $\text{delta}(1) = 1$. The vector *delta* contains the information necessary to construct the (singular) eigenvectors.

sigma REAL for slasd4
DOUBLE PRECISION for dlasd4
The computed *sigma_i*, the *i*-th updated eigenvalue.

info INTEGER.
= 0: successful exit
> 0: If *info* = 1, the updating process failed.

?lasd5

Computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.

Syntax

```
call slasd5( i, d, z, delta, rho, dsigma, work )
```

```
call dlasd5( i, d, z, delta, rho, dsigma, work )
```

Include Files

- mkl.fi

Description

The routine computes the square root of the *i*-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(d) * \text{diag}(d) + \text{rho} * Z * Z^T$

The diagonal entries in the array d must satisfy $0 \leq d(i) < d(j)$ for $i < j$, ρ must be greater than 0, and that the Euclidean norm of the vector Z is equal to 1.

Input Parameters

i	<i>INTEGER</i> . The index of the eigenvalue to be computed. $i = 1$ or $i = 2$.
d	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> Array, dimension (2). The original eigenvalues, $0 \leq d(1) < d(2)$.
z	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> Array, dimension (2). The components of the updating vector.
ρ	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> The scalar in the symmetric updating formula.
$work$	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> . Workspace array, dimension (2). Contains $(d(j) + \sigma_i)$ in its j -th component.

Output Parameters

δ	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> . Array, dimension (2). Contains $(d(j) - \sigma_i)$ in its j -th component. The vector δ contains the information necessary to construct the eigenvectors.
σ_i	REAL for <i>slasd5</i> DOUBLE PRECISION for <i>dlsd5</i> . The computed σ_i , the i -th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

Syntax

```
call slasd6( icompr, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

```
call dlasd6( icompg, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

Include Files

- mkl.fi

Description

The routine `?lasd6` computes the *SVD* of an updated upper bidiagonal matrix B obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. B is an n -by- m matrix with $n = nl + nr + 1$ and $m = n + sqre$. A related subroutine, `?lasd1`, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. `?lasd6` computes the *SVD* as follows:

$$B = U(i_n) \begin{bmatrix} D1(i_n) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(i_n) & 0 \end{bmatrix} * VT(i_n)$$

= $U(out) * (D(out) * VT(out))$

where $Z' = (Z1' \ a \ Z2' \ b) = u' * VT'$, and u is a vector of dimension m with $alpha$ and $beta$ in the $n/1+1$ and $n/1+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The singular values of B can be computed using $D1$, $D2$, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in vf and vl , respectively, in `?lasd6`. Hence U and VT are not explicitly referenced.

The singular values are stored in D . The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the Z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd7`.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine `?lasd4` (as called by `?lasd8`). This routine also updates vf and vl and computes the distances between the updated singular values and the old singular values. `?lasd6` is called from `?lasda`.

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form: = 0: Compute singular values only = 1: Compute singular vectors in factored form as well.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER .

= 0: the lower block is an nr -by- nr square matrix.

= 1: the lower block is an nr -by- $(nr+1)$ rectangular matrix.

The bidiagonal matrix has row dimension $n=nl+nr+1$, and column dimension $m = n + sqre$.

<i>d</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Array, dimension ($nl+nr+1$). On entry $d(1:nl,1:nl)$ contains the singular values of the upper block, and $d(nl+2:n)$ contains the singular values of the lower block.</p>
<i>vf</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Array, dimension (m).</p> <p>On entry, $vf(1:nl+1)$ contains the first components of all right singular vectors of the upper block; and $vf(nl+2:m)$ contains the first components of all right singular vectors of the lower block.</p>
<i>vl</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Array, dimension (m).</p> <p>On entry, $vl(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $vl(nl+2:m)$ contains the last components of all right singular vectors of the lower block.</p>
<i>alpha</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Contains the diagonal element associated with the added row.</p>
<i>beta</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Contains the off-diagonal element associated with the added row.</p>
<i>ldgcol</i>	<p>INTEGER. The leading dimension of the output array <i>givcol</i>, must be at least n.</p>
<i>ldgnum</i>	<p>INTEGER.</p> <p>The leading dimension of the output arrays <i>givnum</i> and <i>poles</i>, must be at least n.</p>
<i>work</i>	<p>REAL for <code>slasd6</code></p> <p>DOUBLE PRECISION for <code>dlasd6</code></p> <p>Workspace array, dimension ($4m$).</p>
<i>iwork</i>	<p>INTEGER.</p> <p>Workspace array, dimension ($3n$).</p>

Output Parameters

<i>d</i>	On exit <i>d</i> (1: <i>n</i>) contains the singular values of the modified matrix.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I))), I = 1, n$.
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I))), I = 1, n$.
<i>idxq</i>	INTEGER. Array, dimension (<i>n</i>). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, <i>d</i> (<i>idxq</i> (<i>i</i> = 1, <i>n</i>)) will be in ascending order.
<i>perm</i>	INTEGER. Array, dimension (<i>n</i>). The permutations (from deflation and sorting) to be applied to each block. Not referenced if <i>icompq</i> = 0.
<i>givptr</i>	INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompq</i> = 0.
<i>givcol</i>	INTEGER. Array, dimension (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>givnum</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>ldgnum</i> , 2). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>poles</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>ldgnum</i> , 2). On exit, <i>poles</i> (1,*) is an array containing the new singular values obtained from solving the secular equation, and <i>poles</i> (2,*) is an array containing the poles in the secular equation. Not referenced if <i>icompq</i> = 0.
<i>difl</i>	REAL for slasd6 DOUBLE PRECISION for dlasd6 Array, dimension (<i>n</i>). On exit, <i>difl</i> (<i>i</i>) is the distance between <i>i</i> -th updated (undeflated) singular value and the <i>i</i> -th (undeflated) old singular value.
<i>difr</i>	REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*ldgnum*, 2) if *icompq* = 1 and dimension (*n*) if *icompq* = 0.

On exit, *difr*(*i*, 1) is the distance between *i*-th updated (undeflated) singular value and the *i*+1-th (undeflated) old singular value. If *icompq* = 1, *difr*(1: *k*, 2) is an array containing the normalizing factors for the right singular vector matrix.

See ?lasd8 for details on *difl* and *difr*.

z

REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (*m*).

The first elements of this array contain the components of the deflation-adjusted updating row vector.

k

INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.

c

REAL for slasd6

DOUBLE PRECISION for dlasd6

c contains garbage if *sqre* = 0 and the C-value of a Givens rotation related to the right null space if *sqre* = 1.

s

REAL for slasd6

DOUBLE PRECISION for dlasd6

s contains garbage if *sqre* = 0 and the S-value of a Givens rotation related to the right null space if *sqre* = 1.

info

INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: if *info* = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

```
call slasd7( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta, dsigma,
            idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s, info )
```

```
call dlasd7( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta, dsigma,
            idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s, info )
```

Include Files

- `mkl.fi`

Description

The routine `?lasd7` merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one. `?lasd7` is called from `?lasd6`.

Input Parameters

<i>icompg</i>	<p>INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows:</p> <p>= 0: Compute singular values only.</p> <p>= 1: Compute singular vectors of upper bidiagonal matrix in compact form.</p>
<i>nl</i>	<p>INTEGER. The row dimension of the upper block.</p> <p>$nl \geq 1$.</p>
<i>nr</i>	<p>INTEGER. The row dimension of the lower block.</p> <p>$nr \geq 1$.</p>
<i>sqre</i>	<p>INTEGER.</p> <p>= 0: the lower block is an nr-by-nr square matrix.</p> <p>= 1: the lower block is an nr-by-$(nr+1)$ rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.</p>
<i>d</i>	<p>REAL for <code>slasd7</code></p> <p>DOUBLE PRECISION for <code>dlasd7</code></p> <p>Array, DIMENSION (n). On entry <i>d</i> contains the singular values of the two submatrices to be combined.</p>
<i>zw</i>	<p>REAL for <code>slasd7</code></p> <p>DOUBLE PRECISION for <code>dlasd7</code></p> <p>Array, DIMENSION (m).</p> <p>Workspace for <i>z</i>.</p>
<i>vf</i>	<p>REAL for <code>slasd7</code></p> <p>DOUBLE PRECISION for <code>dlasd7</code></p> <p>Array, DIMENSION (m). On entry, $vf(1:nl+1)$ contains the first components of all right singular vectors of the upper block; and $vf(nl+2:m)$ contains the first components of all right singular vectors of the lower block.</p>
<i>vfw</i>	<p>REAL for <code>slasd7</code></p> <p>DOUBLE PRECISION for <code>dlasd7</code></p> <p>Array, DIMENSION (m).</p>

	Workspace for \mathbf{v}_f .
\mathbf{v}_l	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsasd7</code> Array, DIMENSION (m). On entry, $\mathbf{v}_l(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $\mathbf{v}_l(nl+2:m)$ contains the last components of all right singular vectors of the lower block.
\mathbf{VLW}	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsasd7</code> Array, DIMENSION (m). Workspace for \mathbf{VL} .
α	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsasd7</code> . Contains the diagonal element associated with the added row.
β	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsasd7</code> Contains the off-diagonal element associated with the added row.
\mathbf{idx}	INTEGER. Workspace array, DIMENSION (n). This will contain the permutation used to sort the contents of d into ascending order.
\mathbf{idxp}	INTEGER. Workspace array, DIMENSION (n). This will contain the permutation used to place deflated values of d at the end of the array.
\mathbf{idxq}	INTEGER. Array, DIMENSION (n). This contains the permutation which separately sorts the two sub-problems in d into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have $nl+1$ added to their values.
\mathbf{ldgcol}	INTEGER. The leading dimension of the output array <code>givcol</code> , must be at least n .
\mathbf{ldgnum}	INTEGER. The leading dimension of the output array <code>givnum</code> , must be at least n .

Output Parameters

k	INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation.
-----	--

	$1 \leq k \leq n$.
<i>d</i>	On exit, <i>d</i> contains the trailing (<i>n-k</i>) updated singular values (those which were deflated) sorted into increasing order.
<i>z</i>	REAL for <i>slasd7</i> DOUBLE PRECISION for <i>dlasd7</i> . Array, DIMENSION (<i>m</i>). On exit, <i>Z</i> contains the updating row vector in the secular equation.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>dsigma</i>	REAL for <i>slasd7</i> DOUBLE PRECISION for <i>dlasd7</i> . Array, DIMENSION (<i>n</i>). Contains a copy of the diagonal elements (<i>k-1</i> singular values and one zero) in the secular equation.
<i>idxp</i>	On output, <i>idxp</i> (2: <i>k</i>) points to the nondeflated <i>d</i> -values and <i>idxp</i> (<i>k+1:n</i>) points to the deflated singular values.
<i>perm</i>	INTEGER. Array, DIMENSION (<i>n</i>). The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if <i>icompg</i> = 0.
<i>givptr</i>	INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompg</i> = 0.
<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompg</i> = 0.
<i>givnum</i>	REAL for <i>slasd7</i> DOUBLE PRECISION for <i>dlasd7</i> . Array, DIMENSION (<i>ldgnum</i> , 2). Each number indicates the <i>C</i> or <i>S</i> value to be used in the corresponding Givens rotation. Not referenced if <i>icompg</i> = 0.
<i>c</i>	REAL for <i>slasd7</i> . DOUBLE PRECISION for <i>dlasd7</i> . If <i>sqre</i> = 0, then <i>c</i> contains garbage, and if <i>sqre</i> = 1, then <i>c</i> contains <i>C</i> -value of a Givens rotation related to the right null space.
<i>S</i>	REAL for <i>slasd7</i> .

DOUBLE PRECISION for `dlasd7`.

If `sqre = 0`, then `s` contains garbage, and if `sqre = 1`, then `s` contains S -value of a Givens rotation related to the right null space.

`info`

INTEGER.

= 0: successful exit.

< 0: if `info = -i`, the i -th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd8( icompg, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

```
call dlasd8( icompg, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

Include Files

- `mkl.fi`

Description

The routine `?lasd8` finds the square roots of the roots of the secular equation, as defined by the values in `dsigma` and `z`. It makes the appropriate calls to `?lasd4`, and stores, for each element in d , the distance to its two nearest poles (elements in `dsigma`). It also updates the arrays `vf` and `vl`, the first and last components of all the right singular vectors of the original bidiagonal matrix. `?lasd8` is called from `?lasd6`.

Input Parameters

<code>icompg</code>	<p>INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:</p> <p>= 0: Compute singular values only.</p> <p>= 1: Compute singular vectors in factored form as well.</p>
<code>k</code>	<p>INTEGER. The number of terms in the rational function to be solved by <code>?lasd4</code>. $k \geq 1$.</p>
<code>z</code>	<p>REAL for <code>slasd8</code></p> <p>DOUBLE PRECISION for <code>dlasd8</code>.</p> <p>Array, DIMENSION (k).</p> <p>The first k elements of this array contain the components of the deflation-adjusted updating row vector.</p>
<code>vf</code>	<p>REAL for <code>slasd8</code></p> <p>DOUBLE PRECISION for <code>dlasd8</code>.</p> <p>Array, DIMENSION (k).</p> <p>On entry, <code>vf</code> contains information passed through <code>dbede8</code>.</p>

<i>vl</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Array, DIMENSION (<i>k</i>). On entry, <i>vl</i> contains information passed through <i>dbede8</i> .
<i>lddifr</i>	INTEGER. The leading dimension of the output array <i>difr</i> , must be at least <i>k</i> .
<i>dsigma</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>work</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Workspace array, DIMENSION at least (3 <i>k</i>).

Output Parameters

<i>d</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Array, DIMENSION (<i>k</i>). On output, <i>D</i> contains the updated singular values.
<i>z</i>	Updated on exit.
<i>vf</i>	On exit, <i>vf</i> contains the first <i>k</i> components of the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the first <i>k</i> components of the last components of all right singular vectors of the bidiagonal matrix.
<i>difl</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Array, DIMENSION (<i>k</i>). On exit, $difl(i) = d(i) - dsigma(i)$.
<i>difr</i>	REAL for <i>slasd8</i> DOUBLE PRECISION for <i>dlasd8</i> . Array, DIMENSION (<i>lddifr</i> , 2) if <i>icompq</i> = 1 and DIMENSION (<i>k</i>) if <i>icompq</i> = 0. On exit, $difr(i,1) = d(i) - dsigma(i+1)$, $difr(k,1)$ is not defined and will not be referenced. If <i>icompq</i> = 1, $difr(1:k,2)$ is an array containing the normalizing factors for the right singular vector matrix.
<i>dsigma</i>	The elements of this array may be very slightly altered in value.

info INTEGER.
 = 0: successful exit.
 < 0: if *info* = $-i$, the i -th argument had an illegal value.
 > 0: If *info* = 1, an singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd9( icoompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9( icoompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Include Files

- mkl.fi

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.

Input Parameters

icoompq INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:
 If *icoompq* = 0, compute singular values only;
 If *icoompq* = 1, compute singular vector matrices in factored form also.

k INTEGER. The number of terms in the rational function to be solved by slasd4. $k \geq 1$.

dsigma REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION(*k*).
 The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION (*k*). The first *k* elements of this array contain the components of the deflation-adjusted updating row vector.

vf REAL for slasd9
 DOUBLE PRECISION for dlasd9.

Array, DIMENSION(k). On entry, vf contains information passed through `sbede8`.

 vl REAL for `slasd9`DOUBLE PRECISION for `dlsd9`.

Array, DIMENSION(k). On entry, vl contains information passed through `sbede8`.

 $work$ REAL for `slasd9`DOUBLE PRECISION for `dlsd9`.

Workspace array, DIMENSION at least $(3k)$.

Output Parameters

 d REAL for `slasd9`DOUBLE PRECISION for `dlsd9`.

Array, DIMENSION(k). $d(i)$ contains the updated singular values.

 vf

On exit, vf contains the first k components of the first components of all right singular vectors of the bidiagonal matrix.

 vl

On exit, vl contains the first k components of the last components of all right singular vectors of the bidiagonal matrix.

 $difl$ REAL for `slasd9`DOUBLE PRECISION for `dlsd9`.

Array, DIMENSION (k).

On exit, $difl(i) = d(i) - dsigma(i)$.

 $difr$ REAL for `slasd9`DOUBLE PRECISION for `dlsd9`.

Array,

DIMENSION ($ldu, 2$) if $icompg = 1$ and

DIMENSION (k) if $icompg = 0$.

On exit, $difr(i, 1) = d(i) - dsigma(i+1)$, $difr(k, 1)$ is not defined and will not be referenced.

If $icompg = 1$, $difr(1:k, 2)$ is an array containing the normalizing factors for the right singular vector matrix.

 $info$

INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

> 0: If $info = 1$, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
call slasda( icompg, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z, poles,
givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

```
call dlasda( icompg, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z, poles,
givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

Include Files

- mkl.fi

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e , where $m = n + sqre$.

The algorithm computes the singular values in the $SVDB = U*S*VT$. The orthogonal matrices U and VT are optionally computed in compact form. A related subroutine ?lasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

<i>icompg</i>	INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only. = 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>smlsiz</i>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	INTEGER. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .
<i>sqre</i>	INTEGER. Specifies the column dimension of the bidiagonal matrix. If $sqre = 0$: the bidiagonal matrix has column dimension $m = n$ If $sqre = 1$: the bidiagonal matrix has column dimension $m = n + 1$.
<i>d</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (n). On entry, d contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for slasda DOUBLE PRECISION for dlasda.

Array, DIMENSION ($m - 1$). Contains the subdiagonal entries of the bidiagonal matrix. On exit, e is destroyed.

ldu INTEGER. The leading dimension of arrays u , vt , $difl$, $difr$, $poles$, $givnum$, and z . $ldu \geq n$.

ldgcol INTEGER. The leading dimension of arrays $givcol$ and $perm$. $ldgcol \geq n$.

work REAL for *slasda*
DOUBLE PRECISION for *dlasda*.
Workspace array, DIMENSION ($6n + (smlsiz + 1)^2$).

iwork INTEGER.
Workspace array, *Dimension* must be at least ($7n$).

Output Parameters

d On exit d , if $info = 0$, contains the singular values of the bidiagonal matrix.

u REAL for *slasda*
DOUBLE PRECISION for *dlasda*.
Array, DIMENSION (ldu , $smlsiz$) if $icompg = 1$.
Not referenced if $icompg = 0$.
If $icompg = 1$, on exit, u contains the left singular vector matrices of all subproblems at the bottom level.

vt REAL for *slasda*
DOUBLE PRECISION for *dlasda*.
Array, DIMENSION (ldu , $smlsiz + 1$) if $icompg = 1$, and not referenced if $icompg = 0$. If $icompg = 1$, on exit, vt' contains the right singular vector matrices of all subproblems at the bottom level.

k INTEGER.
Array, DIMENSION (n) if $icompg = 1$ and
DIMENSION (1) if $icompg = 0$.
If $icompg = 1$, on exit, $k(i)$ is the dimension of the i -th secular equation on the computation tree.

difl REAL for *slasda*
DOUBLE PRECISION for *dlasda*.
Array, DIMENSION (ldu , $nlvl$),
where $nlvl = \text{floor}(\log_2(n/smlsiz))$.

difr REAL for *slasda*
DOUBLE PRECISION for *dlasda*.
Array,

DIMENSION (*ldu*, 2 *nlvl*) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0.

If *icompq* = 1, on exit, *difl*(1:*n*, *i*) and *difr*(1:*n*, 2*i* - 1) record distances between singular values on the *i*-th level and singular values on the (*i* - 1)-th level, and *difr*(1:*n*, 2*i*) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z

REAL for *slasda*

DOUBLE PRECISION for *dlasda*.

Array,

DIMENSION (*ldu*, *nlvl*) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0. The first *k* elements of *z*(1, *i*) contain the components of the deflation-adjusted updating row vector for subproblems on the *i*-th level.

poles

REAL for *slasda*

DOUBLE PRECISION for *dlasda*

Array, DIMENSION(*ldu*, 2**nlvl*)

if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *poles*(1, 2*i* - 1) and *poles*(1, 2*i*) contain the new and old singular values involved in the secular equations on the *i*-th level.

givptr

INTEGER. Array, DIMENSION (*n*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol

INTEGER .

Array, DIMENSION(*ldgcol*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givcol*(1, 2*i* - 1) and *givcol*(1, 2*i*) record the locations of Givens rotations performed on the *i*-th level on the computation tree.

perm

INTEGER . Array, DIMENSION (*ldgcol*, *nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *perm*(1, *i*) records permutations done on the *i*-th level of the computation tree.

givnum

REAL for *slasda*

DOUBLE PRECISION for *dlasda*.

Array DIMENSION (*ldu*, 2**nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, 2*i* - 1) and *givnum*(1, 2*i*) record the C- and S-values of Givens rotations performed on the *i*-th level on the computation tree.

c

REAL for *slasda*

DOUBLE PRECISION for *dlasda*.

Array,

DIMENSION (*n*) if *icompq* = 1, and

DIMENSION (1) if $icompg = 0$.

If $icompg = 1$ and the i -th subproblem is not square, on exit, $c(i)$ contains the C -value of a Givens rotation related to the right null space of the i -th subproblem.

s

REAL for `slasda`

DOUBLE PRECISION for `dlasda`.

Array,

DIMENSION (n) $icompg = 1$, and

DIMENSION (1) if $icompg = 0$.

If $icompg = 1$ and the i -th subproblem is not square, on exit, $s(i)$ contains the S -value of a Givens rotation related to the right null space of the i -th subproblem.

$info$

INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value

> 0: If $info = 1$, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by `?bdsdc`.

Syntax

```
call slasdq( uplo, sqre, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

```
call dlasdq( uplo, sqre, n, ncv, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

Include Files

- `mkl.fi`

Description

The routine `?lasdq` computes the singular value decomposition (SVD) of a real (upper or lower) bidiagonal matrix with diagonal d and off-diagonal e , accumulating the transformations if desired. If B is the input bidiagonal matrix, the algorithm computes orthogonal matrices Q and P such that $B = Q^* S^* P^T$. The singular values S are overwritten on d .

The input matrix U is changed to $U^* Q$ if desired.

The input matrix VT is changed to $P^T * VT$ if desired.

The input matrix C is changed to $Q^T * C$ if desired.

Input Parameters

$uplo$

CHARACTER*1. On entry, $uplo$ specifies whether the input bidiagonal matrix is upper or lower bidiagonal.

If $uplo = 'U'$ or $'u'$, B is upper bidiagonal;

If $uplo = 'L'$ or $'l'$, B is lower bidiagonal.

sqr

INTEGER.

= 0: then the input matrix is n -by- n .

= 1: then the input matrix is n -by- $(n+1)$ if $uplu = 'U'$ and $(n+1)$ -by- n if $uplu$

= $'L'$. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqr \geq n$ columns.

n

INTEGER. On entry, n specifies the number of rows and columns in the matrix. n must be at least 0.

$ncvt$

INTEGER. On entry, $ncvt$ specifies the number of columns of the matrix VT . $ncvt$ must be at least 0.

nru

INTEGER. On entry, nru specifies the number of rows of the matrix U . nru must be at least 0.

ncc

INTEGER. On entry, ncc specifies the number of columns of the matrix C . ncc must be at least 0.

d

REAL for *slasdq*

DOUBLE PRECISION for *dlasdq*.

Array, DIMENSION (n). On entry, d contains the diagonal entries of the bidiagonal matrix.

e

REAL for *slasdq*

DOUBLE PRECISION for *dlasdq*.

Array, DIMENSION is $(n-1)$ if $sqr = 0$ and n if $sqr = 1$. On entry, the entries of e contain the off-diagonal entries of the bidiagonal matrix.

vt

REAL for *slasdq*

DOUBLE PRECISION for *dlasdq*.

Array, DIMENSION ($ldvt$, $ncvt$). On entry, contains a matrix which on exit has been premultiplied by P^T , dimension n -by- $ncvt$ if $sqr = 0$ and $(n+1)$ -by- $ncvt$ if $sqr = 1$ (not referenced if $ncvt=0$).

$ldvt$

INTEGER. On entry, $ldvt$ specifies the leading dimension of vt as declared in the calling (sub) program. $ldvt$ must be at least 1. If $ncvt$ is nonzero, $ldvt$ must also be at least n .

u

REAL for *slasdq*

DOUBLE PRECISION for *dlasdq*.

Array, DIMENSION (ldu , n). On entry, contains a matrix which on exit has been postmultiplied by Q , dimension nru -by- n if $sqr = 0$ and nru -by- $(n+1)$ if $sqr = 1$ (not referenced if $nru=0$).

<i>ldu</i>	INTEGER. On entry, <i>ldu</i> specifies the leading dimension of <i>u</i> as declared in the calling (sub) program. <i>ldu</i> must be at least $\max(1, nru)$.
<i>c</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION (<i>ldc</i> , <i>ncc</i>). On entry, contains an <i>n</i> -by- <i>ncc</i> matrix which on exit has been premultiplied by Q' , dimension <i>n</i> -by- <i>ncc</i> if <i>sqre</i> = 0 and (<i>n</i> +1)-by- <i>ncc</i> if <i>sqre</i> = 1 (not referenced if <i>ncc</i> =0).
<i>ldc</i>	INTEGER. On entry, <i>ldc</i> specifies the leading dimension of <i>C</i> as declared in the calling (sub) program. <i>ldc</i> must be at least 1. If <i>ncc</i> is non-zero, <i>ldc</i> must also be at least <i>n</i> .
<i>work</i>	REAL for <i>slasdq</i> DOUBLE PRECISION for <i>dlasdq</i> . Array, DIMENSION ($4n$). This is a workspace array. Only referenced if one of <i>ncvt</i> , <i>nru</i> , or <i>ncc</i> is nonzero, and if <i>n</i> is at least 2.

Output Parameters

<i>d</i>	On normal exit, <i>d</i> contains the singular values in ascending order.
<i>e</i>	On normal exit, <i>e</i> will contain 0. If the algorithm does not converge, <i>d</i> and <i>e</i> will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.
<i>vt</i>	On exit, the matrix has been premultiplied by P' .
<i>u</i>	On exit, the matrix has been postmultiplied by Q .
<i>c</i>	On exit, the matrix has been premultiplied by Q' .
<i>info</i>	INTEGER. On exit, a value of 0 indicates a successful exit. If <i>info</i> < 0, argument number <i>-info</i> is illegal. If <i>info</i> > 0, the algorithm did not converge, and <i>info</i> specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
call slasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

```
call dlasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Include Files

- mkl.fi

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

<i>n</i>	INTEGER. On entry, the number of diagonal elements of the bidiagonal matrix.
<i>msub</i>	INTEGER. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

<i>lvl</i>	INTEGER. On exit, the number of levels on the computation tree.
<i>nd</i>	INTEGER. On exit, the number of nodes on the tree.
<i>inode</i>	INTEGER. Array, DIMENSION (<i>n</i>). On exit, centers of subproblems.
<i>ndiml</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of left children.
<i>ndimr</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset( uplo, m, n, alpha, beta, a, lda )
call dlaset( uplo, m, n, alpha, beta, a, lda )
call claset( uplo, m, n, alpha, beta, a, lda )
call zlaset( uplo, m, n, alpha, beta, a, lda )
```

Include Files

- mkl.fi

Description

The routine initializes an *m*-by-*n* matrix *A* to *beta* on the diagonal and *alpha* on the off-diagonals.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix <i>A</i> to be set. If <i>uplo</i> = 'U', upper triangular part is set; the strictly lower triangular part of <i>A</i> is not changed. If <i>uplo</i> = 'L': lower triangular part is set; the strictly upper triangular part of <i>A</i> is not changed.
-------------	---

	Otherwise: All of the matrix A is set.
m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
$alpha, beta$	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. The constants to which the off-diagonal and diagonal elements are to be set, respectively.
a	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. Array, DIMENSION (lda, n). The array a contains the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.

Output Parameters

a	On exit, the leading m -by- n submatrix of A is set as follows: if $uplo = 'U'$, $A_{ij} = alpha, 1 \leq i \leq j-1, 1 \leq j \leq n$, if $uplo = 'L'$, $A_{ij} = alpha, j+1 \leq i \leq m, 1 \leq j \leq n$, otherwise, $A_{ij} = alpha, 1 \leq i \leq m, 1 \leq j \leq n, i \neq j$, and, for all $uplo$, $A_{ij} = beta, 1 \leq i \leq \min(m, n)$.
-----	---

?lasq1

Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.

Syntax

```
call slasq1( n, d, e, work, info )
call dlasq1( n, d, e, work, info )
```

Include Files

- mkl.fi

Description

The routine `?lasq1` computes the singular values of a real n -by- n bidiagonal matrix Z with diagonal d and off-diagonal e . The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

<i>n</i>	INTEGER. The number of rows and columns in the matrix. $n \geq 0$.
<i>d</i>	REAL for <code>slasq1</code> DOUBLE PRECISION for <code>dlasq1</code> . Array, DIMENSION (n). On entry, <i>d</i> contains the diagonal elements of the bidiagonal matrix whose SVD is desired.
<i>e</i>	REAL for <code>slasq1</code> DOUBLE PRECISION for <code>dlasq1</code> . Array, DIMENSION (n). On entry, elements $e(1:n-1)$ contain the off-diagonal elements of the bidiagonal matrix whose SVD is desired.
<i>work</i>	REAL for <code>slasq1</code> DOUBLE PRECISION for <code>dlasq1</code> . Workspace array, DIMENSION ($4n$).

Output Parameters

<i>d</i>	On normal exit, <i>d</i> contains the singular values in decreasing order.
<i>e</i>	On exit, <i>e</i> is overwritten.
<i>info</i>	INTEGER. = 0: successful exit; < 0: if $info = -i$, the i -th argument had an illegal value; > 0: the algorithm failed: = 1, a split was marked by a positive value in <i>e</i> ; = 2, current block of Z not diagonalized after $100n$ iterations (in inner while loop) - on exit the current contents of <i>d</i> and <i>e</i> represent a matrix with the same singular values as the matrix with which <code>?lasq1</code> was originally called, and which the calling subroutine could use to finish the computation, or even feed back into <code>?lasq1</code> ; = 3, termination criterion of outer while loop not met (program created more than n unreduced blocks).

?lasq2

Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array z to high relative accuracy.

Used by ?bdsqr and ?stegr.

Syntax

```
call slasq2( n, z, info )
```

```
call dlasq2( n, z, info )
```

Include Files

- mkl.fi

Description

The routine ?lasq2 computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array z to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of z to the tridiagonal matrix, let L be a unit lower bidiagonal matrix with subdiagonals $z(2,4,6,\dots)$ and let U be an upper bidiagonal matrix with 1's above and diagonal $z(1,3,5,\dots)$. The tridiagonal is LU or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

n INTEGER. The number of rows and columns in the matrix. $n \geq 0$.

z REAL for slasq2
DOUBLE PRECISION for dlasq2.
Array, DIMENSION (4 * n).
On entry, z holds the quotient difference array.

Output Parameters

z On exit, entries 1 to n hold the eigenvalues in decreasing order, $z(2n+1)$ holds the trace, and $z(2n+2)$ holds the sum of the eigenvalues. If $n > 2$, then $z(2n+3)$ holds the iteration count, $z(2n+4)$ holds $n\text{divs}/n^2$, and $z(2n+5)$ holds the percentage of shifts that failed.

$info$ INTEGER.
= 0: successful exit;
< 0: if the i -th argument is a scalar and had an illegal value, then $info = -i$, if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100 + j)$;
> 0: the algorithm failed:
= 1, a split was marked by a positive value in e ;
= 2, current block of z not diagonalized after $100*n$ iterations (in inner while loop) - On exit z holds a quotient difference array with the same eigenvalues as the z array on entry;

= 3, termination criterion of outer while loop not met (program created more than n unreduced blocks).

Application Notes

The routine `?lasq2` defines a logical variable, `ieee`, which is `.TRUE.` on machines which follow ieee-754 floating-point standard in their handling of infinities and NaNs, and `.FALSE.` otherwise. This variable is passed to `?lasq3`.

?lasq3

Checks for deflation, computes a shift and calls dqds.
Used by ?bdsqr.

Syntax

```
call slasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee, ttype,
dmin1, dmin2, dn, dn1, dn2, g, tau )
```

```
call dlasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee, ttype,
dmin1, dmin2, dn, dn1, dn2, g, tau )
```

Include Files

- `mkl.fi`

Description

The routine `?lasq3` checks for deflation, computes a shift `tau`, and calls `dqds`. In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

<code>i0</code>	INTEGER. First index.
<code>n0</code>	INTEGER. Last index.
<code>z</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Array, DIMENSION (4n). <code>z</code> holds the <code>qd</code> array.
<code>pp</code>	INTEGER. <code>pp=0</code> for ping, <code>pp=1</code> for pong. <code>pp=2</code> indicates that flipping was applied to the <code>Z</code> array and that the initial tests for deflation should not be performed.
<code>desig</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Lower order part of <code>sigma</code> .
<code>qmax</code>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Maximum value of <code>q</code> .
<code>ieee</code>	LOGICAL.

Flag for `ieee` or non-`ieee` arithmetic (passed to `?lasq5`).

`ttype` INTEGER.
Shift type.

`dmin1, dmin2, dn, dn1, dn2, g, tau` REAL for `slasq3`
DOUBLE PRECISION for `dlasq3`.

These scalars are passed as arguments in order to save their values between calls to `?lasq3`.

Output Parameters

`dmin` REAL for `slasq3`
DOUBLE PRECISION for `dlasq3`.
Minimum value of d .

`pp` INTEGER. $pp=0$ for ping, $pp=1$ for pong. $pp=2$ indicates that flipping was applied to the Z array and that the initial tests for deflation should not be performed.

`sigma` REAL for `slasq3`
DOUBLE PRECISION for `dlasq3`.
Sum of shifts used in the current segment.

`desig` Lower order part of σ .

`nfail` INTEGER. Number of times shift was too big.

`iter` INTEGER. Number of iterations.

`ndiv` INTEGER. Number of divisions.

`ttype` INTEGER.
Shift type.

`dmin1, dmin2, dn, dn1, dn2, g, tau` REAL for `slasq3`
DOUBLE PRECISION for `dlasq3`.

These scalars are passed as arguments in order to save their values between calls to `?lasq3`.

?lasq4

Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by `?bdsqr`.

Syntax

```
call slasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
call dlasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
```

Include Files

- `mkl.fi`

Description

The routine computes an approximation τ to the smallest eigenvalue using values of d from the previous transform.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the <i>qd</i> array.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>n0in</i>	INTEGER. The value of <i>n0</i> at start of <code>eigtst</code> .
<i>dmin</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Minimum value of <i>d</i> .
<i>dmin1</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>).
<i>dmin2</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Minimum value of <i>d</i> , excluding <i>d</i> (<i>n0</i>) and <i>d</i> (<i>n0</i> -1).
<i>dn</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Contains <i>d</i> (<i>n</i>).
<i>dn1</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Contains <i>d</i> (<i>n</i> -1).
<i>dn2</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Contains <i>d</i> (<i>n</i> -2).
<i>g</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . A scalar passed as an argument in order to save its value between calls to ? <code>lasq4</code> .

Output Parameters

<i>tau</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. Shift.
<i>ttype</i>	INTEGER. Shift type.
<i>g</i>	REAL for slasq4 DOUBLE PRECISION for dlasq4. A scalar passed as an argument in order to save its value between calls to ?lasq4.

?lasq5

Computes one dqds transform in ping-pong form.
Used by ?bdsqr and ?stegr.

Syntax

```
call slasq5( i0, n0, z, pp, tau, sigma, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee, eps )
```

```
call dlasq5( i0, n0, z, pp, tau, sigma, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee, eps )
```

Include Files

- mkl.fi

Description

The routine computes one dqds transform in ping-pong form: one version for ieee machines, another for non-ieee machines.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. Array, DIMENSION (4 <i>n</i>). <i>z</i> holds the qd array. <i>emin</i> is stored in <i>z</i> (4* <i>n0</i>) to avoid an extra argument.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>tau</i>	REAL for slasq5 DOUBLE PRECISION for dlasq5. This is the shift.

sigma REAL for slasq5
DOUBLE PRECISION for dlasq5.
This is the accumulated shift up to the current point.

ieee LOGICAL. Flag for IEEE or non-IEEE arithmetic.

eps REAL for slasq5
DOUBLE PRECISION for dlasq5.
This is the value of epsilon used.

Output Parameters

dmin REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of d .

dmin1 REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of d , excluding $d(n0)$.

dmin2 REAL for slasq5
DOUBLE PRECISION for dlasq5.
Minimum value of d , excluding $d(n0)$ and $d(n0-1)$.

dn REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains $d(n0)$, the last value of d .

dnm1 REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains $d(n0-1)$.

dnm2 REAL for slasq5
DOUBLE PRECISION for dlasq5. Contains $d(n0-2)$.

?lasq6

Computes one dqd transform in ping-pong form. Used by ?bdsqr and ?stegr.

Syntax

```
call slasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Include Files

- mkl.fi

Description

The routine `?lasq6` computes one *dqd* (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

i0 INTEGER. First index.

n0 INTEGER. Last index.

z REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`.
Array, DIMENSION (4*n*). *Z* holds the *qd* array. *emin* is stored in *z*(4**n0*) to avoid an extra argument.

pp INTEGER. *pp*=0 for ping, *pp*=1 for pong.

Output Parameters

dmin REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`.
Minimum value of *d*.

dmin1 REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`.
Minimum value of *d*, excluding *d*(*n0*).

dmin2 REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`.
Minimum value of *d*, excluding *d*(*n0*) and *d*(*n0*-1).

dn REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`. Contains *d*(*n0*), the last value of *d*.

dnm1 REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`. Contains *d*(*n0*-1).

dnm2 REAL for `slasq6`
DOUBLE PRECISION for `dlasq6`. Contains *d*(*n0*-2).

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr( side, pivot, direct, m, n, c, s, a, lda )
call dlasr( side, pivot, direct, m, n, c, s, a, lda )
call clasr( side, pivot, direct, m, n, c, s, a, lda )
```


<i>pivot</i>	<p>CHARACTER*1. Specifies the plane for which $P(k)$ is a plane rotation matrix.</p> <p>= 'V': Variable pivot, the plane $(k, k+1)$</p> <p>= 'T': Top pivot, the plane $(1, k+1)$</p> <p>= 'B': Bottom pivot, the plane (k, z)</p>
<i>m</i>	<p>INTEGER. The number of rows of the matrix A.</p> <p>If $m \leq 1$, an immediate return is effected.</p>
<i>n</i>	<p>INTEGER. The number of columns of the matrix A.</p> <p>If $n \leq 1$, an immediate return is effected.</p>
<i>c, s</i>	<p>REAL for <code>slasr/clasr</code></p> <p>DOUBLE PRECISION for <code>dlasr/zlasr</code>.</p> <p>Arrays, DIMENSION</p> <p>$(m-1)$ if <i>side</i> = 'L',</p> <p>$(n-1)$ if <i>side</i> = 'R' .</p> <p>$c(k)$ and $s(k)$ contain the cosine and sine of the plane rotations respectively that define the 2-by-2 plane rotation part $(R(k))$ of the $P(k)$ matrix as described above in <i>Description</i>.</p>
<i>a</i>	<p>REAL for <code>slasr</code></p> <p>DOUBLE PRECISION for <code>dlasr</code></p> <p>COMPLEX for <code>clasr</code></p> <p>DOUBLE COMPLEX for <code>zlasr</code>.</p> <p>Array, DIMENSION (lda, n).</p> <p>The m-by-n matrix A.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array a.</p> <p>$lda \geq \max(1, m)$.</p>

Output Parameters

<i>a</i>	On exit, A is overwritten by $P*A$ if <i>side</i> = 'R', or by $A*P$ if <i>side</i> = 'L'.
----------	--

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt( id, n, d, info )
```

```
call dlasrt( id, n, d, info )
```

Include Files

- `mkl.fi`

Description

The routine `?lasrt` sorts the numbers in d in increasing order (if $id = 'I'$) or in decreasing order (if $id = 'D'$). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of `stack` limits n to about 2^{32} .

Input Parameters

The data types are given for the Fortran interface.

id	CHARACTER*1. ($d(1) \leq \dots \leq d(n)$) or into decreasing order ($d(1) \geq \dots \geq d(n)$), depending on id .
n	INTEGER. The length of the array d .
d	REAL for <code>slasrt</code> DOUBLE PRECISION for <code>dlasrt</code> . On entry, the array to be sorted.

Output Parameters

d	On exit, d has been sorted into increasing order ($d[0] \leq d[1] \leq \dots \leq d[n-1]$) or into decreasing order ($d[0] \geq d[1] \geq \dots \geq d[n-1]$), depending on id .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info < 0$, the i -th parameter had an illegal value.

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slasq( n, x, incx, scale, sumsq )
call dlassq( n, x, incx, scale, sumsq )
call classq( n, x, incx, scale, sumsq )
call zlassq( n, x, incx, scale, sumsq )
```

Include Files

- `mkl.fi`

Description

The real routines `slasq/dlassq` return the values scl and $smsq$ such that

$$scl^2 * smsq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = x(1 + (i - 1) incx)$.

The value of $sumsq$ is assumed to be non-negative and scl returns the value

$$scl = \max(scale, \text{abs}(x(i))).$$

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *smsq* are overwritten on *scale* and *sumsq*, respectively.

The complex routines *classq/zlassq* return the values *scl* and *ssq* such that

$$scl^2 * ssq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{abs}(x(1 + (i - 1) * incx))$.

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \leq ssq \leq sumsq + 2n$

scale is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{aimag}(x(i))))).$$

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *ssq* are overwritten on *scale* and *sumsq*, respectively.

All routines *?lassq* make only one pass through the vector *x*.

Input Parameters

<i>n</i>	INTEGER. The number of elements to be used from the vector <i>x</i> .
<i>x</i>	REAL for <i>slassq</i> DOUBLE PRECISION for <i>dlassq</i> COMPLEX for <i>classq</i> DOUBLE COMPLEX for <i>zlassq</i> . The vector for which a scaled sum of squares is computed: $x(i) = x(1 + (i-1) * incx), 1 \leq i \leq n$.
<i>incx</i>	INTEGER. The increment between successive values of the vector <i>x</i> . <i>incx</i> > 0.
<i>scale</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>scale</i> in the equation above.
<i>sumsq</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>sumsq</i> in the equation above.

Output Parameters

<i>scale</i>	On exit, <i>scale</i> is overwritten with <i>scl</i> , the scaling factor for the sum of squares.
<i>sumsq</i>	For real flavors: On exit, <i>sumsq</i> is overwritten with the value <i>smsq</i> in the equation above. For complex flavors: On exit, <i>sumsq</i> is overwritten with the value <i>ssq</i> in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

```
call slasv2( f, g, h, ssmi, ssmx, snr, csr, snl, csl )
```

```
call dlasv2( f, g, h, ssmi, ssmx, snr, csr, snl, csl )
```

Include Files

- mkl.fi

Description

The routine ?lasv2 computes the singular value decomposition of a 2-by-2 triangular matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, $\text{abs}(ssmax)$ is the larger singular value, $\text{abs}(ssmin)$ is the smaller singular value, and (csl, snl) and (csr, snr) are the left and right singular vectors for $\text{abs}(ssmax)$, giving the decomposition

$$\begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix} = \begin{bmatrix} ssmx & 0 \\ 0 & ssmin \end{bmatrix}$$

Input Parameters

f, g, h REAL for slasv2
DOUBLE PRECISION for dlasv2.
The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, ssmx REAL for slasv2
DOUBLE PRECISION for dlasv2.
 $\text{abs}(ssmin)$ and $\text{abs}(ssmx)$ is the smaller and the larger singular value, respectively.

snl, csl REAL for slasv2
DOUBLE PRECISION for dlasv2.
The vector (csl, snl) is a unit left singular vector for the singular value $\text{abs}(ssmx)$.

snr, csr REAL for slasv2
DOUBLE PRECISION for dlasv2.

The vector (*csr*, *snr*) is a unit right singular vector for the singular value $\text{abs}(ssmax)$.

Application Notes

Any input parameter may be aliased with any output parameter.

Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulps).

In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp( n, a, lda, k1, k2, ipiv, incx )
call dlaswp( n, a, lda, k1, k2, ipiv, incx )
call claswp( n, a, lda, k1, k2, ipiv, incx )
call zlaswp( n, a, lda, k1, k2, ipiv, incx )
```

Include Files

- mkl.fi

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

The data types are given for the Fortran interface.

<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> .
<i>a</i>	REAL for slaswp DOUBLE PRECISION for dlaswp COMPLEX for claswp DOUBLE COMPLEX for zlaswp. Array, size <i>lda</i> by <i>n</i> . Array <i>a</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> .
<i>k1</i>	INTEGER. The first element of <i>ipiv</i> for which a row interchange will be done.

k2 INTEGER. The last element of *ipiv* for which a row interchange will be done.

ipiv INTEGER.
 Array, size $k1 + (k2 - k1) * |incx|$.
 The vector of pivot indices. Only the elements in positions *k1* through *k2* of *ipiv* are accessed.
ipiv(*k*) = 1 implies rows *k* and *l* are to be interchanged.

incx INTEGER. The increment between successive values of *ipiv*. If *ipiv* is negative, the pivots are applied in reverse order.

Output Parameters

a On exit, the permuted matrix.

?slasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx,
xnorm, info )
```

```
call dslasy2( ltranl, ltranr, isgn, n1, n2, tl, ldtl, tr, ldtr, b, ldb, scale, x, ldx,
xnorm, info )
```

Include Files

- mkl.fi

Description

The routine solves for the *n1*-by-*n2* matrix *X*, $1 \leq n1, n2 \leq 2$, in

$$\text{op}(TL) * X + \text{isgn} * X * \text{op}(TR) = \text{scale} * B,$$

where

TL is *n1*-by-*n1*,

TR is *n2*-by-*n2*,

B is *n1*-by-*n2*,

and *isgn* = 1 or -1. $\text{op}(T) = T$ or T^T , where T^T denotes the transpose of *T*.

Input Parameters

ltranl LOGICAL.
 On entry, *ltranl* specifies the $\text{op}(TL)$:
 = .FALSE., $\text{op}(TL) = TL$,
 = .TRUE., $\text{op}(TL) = (TL)^T$.

ltranr LOGICAL.

On entry, *ltranr* specifies the $op(TR)$:

= `.FALSE.`, $op(TR) = TR$,
 = `.TRUE.`, $op(TR) = (TR)^T$.

isgn INTEGER. On entry, *isgn* specifies the sign of the equation as described before. *isgn* may only be 1 or -1.

n1 INTEGER. On entry, *n1* specifies the order of matrix *TL*.
n1 may only be 0, 1 or 2.

n2 INTEGER. On entry, *n2* specifies the order of matrix *TR*.
n2 may only be 0, 1 or 2.

t1 REAL for `slasy2`
 DOUBLE PRECISION for `dlasy2`.
 Array, DIMENSION (*ldt1*,2).
 On entry, *t1* contains an *n1*-by-*n1* matrix *TL*.

ldt1 INTEGER. The leading dimension of the matrix *TL*.
ldt1 ≥ max(1, *n1*).

tr REAL for `slasy2`
 DOUBLE PRECISION for `dlasy2`.
 Array, DIMENSION (*ldtr*,2). On entry, *tr* contains an *n2*-by-*n2* matrix *TR*.

ldtr INTEGER. The leading dimension of the matrix *TR*.
ldtr ≥ max(1, *n2*).

b REAL for `slasy2`
 DOUBLE PRECISION for `dlasy2`.
 Array, DIMENSION (*ldb*,2). On entry, the *n1*-by-*n2* matrix *B* contains the right-hand side of the equation.

ldb INTEGER. The leading dimension of the matrix *B*.
ldb ≥ max(1, *n1*).

ldx INTEGER. The leading dimension of the output matrix *X*.
ldx ≥ max(1, *n1*).

Output Parameters

scale REAL for `slasy2`
 DOUBLE PRECISION for `dlasy2`.
 On exit, *scale* contains the scale factor.
scale is chosen less than or equal to 1 to prevent the solution overflowing.

<i>x</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. Array, DIMENSION (<i>ldx</i> ,2). On exit, <i>x</i> contains the <i>n1</i> -by- <i>n2</i> solution.
<i>xnorm</i>	REAL for slasy2 DOUBLE PRECISION for dlasy2. On exit, <i>xnorm</i> is the infinity-norm of the solution.
<i>info</i>	INTEGER. On exit, <i>info</i> is set to 0: successful exit. 1: <i>TL</i> and <i>TR</i> have too close eigenvalues, so <i>TL</i> or <i>TR</i> is perturbed to get a nonsingular equation.

NOTE

For higher speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine *?lasyf* computes a partial factorization of a symmetric matrix *A* using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If *uplo* = 'U':

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix}$$

uplo = 'L'

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & I \end{bmatrix}$$

where the order of *D* is at most *nb*.

The actual order is returned in the argument *kb*, and is either *nb* or *nb*-1, or *n* if *n* ≤ *nb*.

This is an auxiliary routine called by `?sytrf`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if `uplo = 'U'`) or A_{22} (if `uplo = 'L'`).

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$.
<code>nb</code>	INTEGER. The maximum number of columns of the matrix A that should be factored. <code>nb</code> should be at least 2 to allow for 2-by-2 pivot blocks.
<code>a</code>	REAL for <code>slasyf</code> DOUBLE PRECISION for <code>dlasyf</code> COMPLEX for <code>clasyf</code> DOUBLE COMPLEX for <code>zlasyf</code> . Array, DIMENSION (<code>lda</code> , <code>n</code>). If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of <code>a</code> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <code>a</code> is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of <code>a</code> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <code>a</code> is not referenced.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $lda \geq \max(1, n)$.
<code>w</code>	REAL for <code>slasyf</code> DOUBLE PRECISION for <code>dlasyf</code> COMPLEX for <code>clasyf</code> DOUBLE COMPLEX for <code>zlasyf</code> . Workspace array, DIMENSION (<code>ldw</code> , <code>nb</code>).
<code>ldw</code>	INTEGER. The leading dimension of the array <code>w</code> . $ldw \geq \max(1, n)$.

Output Parameters

<code>kb</code>	INTEGER. The number of columns of A that were actually factored <code>kb</code> is either <code>nb-1</code> or <code>nb</code> , or <code>n</code> if $n \leq nb$.
<code>a</code>	On exit, <code>a</code> contains details of the partial factorization.
<code>ipiv</code>	INTEGER. Array, DIMENSION (<code>n</code>). Details of the interchanges and the block structure of D . If <code>uplo = 'U'</code> , only the last <code>kb</code> elements of <code>ipiv</code> are set; if <code>uplo = 'L'</code> , only the first <code>kb</code> elements are set.

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D(k, k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?lasyf_rook

Computes a partial factorization of a complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call slasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine `?lasyf_rook` computes a partial factorization of a complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & I \end{bmatrix}$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

This is an auxiliary routine called by `?sytrf_rook`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>nb</i>	INTEGER. The maximum number of columns of the matrix <i>A</i> that should be factored. <i>nb</i> should be at least 2 to allow for 2-by-2 pivot blocks.
<i>a</i>	REAL for slasyf_rook DOUBLE PRECISION for dlasyf_rook COMPLEX for clasyf_rook DOUBLE COMPLEX for zlasyf_rook. Array, DIMENSION (<i>lda</i> , <i>n</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>w</i>	REAL for slasyf_rook DOUBLE PRECISION for dlasyf_rook COMPLEX for clasyf_rook DOUBLE COMPLEX for zlasyf_rook. Workspace array, DIMENSION (<i>ldw</i> , <i>nb</i>).
<i>ldw</i>	INTEGER. The leading dimension of the array <i>w</i> . $ldw \geq \max(1, n)$.

Output Parameters

<i>kb</i>	INTEGER. The number of columns of <i>A</i> that were actually factored <i>kb</i> is either <i>nb</i> -1 or <i>nb</i> , or <i>n</i> if $n \leq nb$.
<i>a</i>	On exit, <i>a</i> contains details of the partial factorization.
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). Details of the interchanges and the block structure of <i>D</i> . If <i>uplo</i> = 'U', only the last <i>kb</i> elements of <i>ipiv</i> are set; if <i>uplo</i> = 'L', only the first <i>kb</i> elements are set. If $ipiv(k) > 0$, then rows and columns <i>k</i> and <i>ipiv</i> (<i>k</i>) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

```
call zlahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine ?lahef computes a partial factorization of a complex Hermitian matrix A , using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If $uplo = 'U'$:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^H & U_{22}^H \end{bmatrix}$$

If $uplo = 'L'$:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^H & L_{21}^H \\ 0 & I \end{bmatrix}$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

Note that U^H denotes the conjugate transpose of U .

This is an auxiliary routine called by ?hetrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored:</p> <p>= 'U': upper triangular</p> <p>= 'L': lower triangular</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>. $n \geq 0$.</p>
<i>nb</i>	<p>INTEGER. The maximum number of columns of the matrix <i>A</i> that should be factored. <i>nb</i> should be at least 2 to allow for 2-by-2 pivot blocks.</p>
<i>a</i>	<p>COMPLEX for <code>clahef</code></p> <p>DOUBLE COMPLEX for <code>zlahef</code>.</p> <p>Array, DIMENSION (<i>lda</i>, <i>n</i>).</p> <p>On entry, the Hermitian matrix <i>A</i>.</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>A</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>A</i> is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>A</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>A</i> is not referenced.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. $lda \geq \max(1, n)$.</p>
<i>w</i>	<p>COMPLEX for <code>clahef</code></p> <p>DOUBLE COMPLEX for <code>zlahef</code>.</p> <p>Workspace array, DIMENSION (<i>ldw</i>, <i>nb</i>).</p>
<i>ldw</i>	<p>INTEGER. The leading dimension of the array <i>w</i>. $ldw \geq \max(1, n)$.</p>

Output Parameters

<i>kb</i>	<p>INTEGER. The number of columns of <i>A</i> that were actually factored <i>kb</i> is either <i>nb</i>-1 or <i>nb</i>, or <i>n</i> if $n \leq nb$.</p>
<i>a</i>	<p>On exit, <i>A</i> contains details of the partial factorization.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>). Details of the interchanges and the block structure of <i>D</i>.</p> <p>If <i>uplo</i> = 'U', only the last <i>kb</i> elements of <i>ipiv</i> are set;</p> <p>if <i>uplo</i> = 'L', only the first <i>kb</i> elements are set.</p> <p>If <i>ipiv</i>(<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i>(<i>k</i>) are interchanged and <i>D</i>(<i>k</i>, <i>k</i>) is a 1-by-1 diagonal block.</p>

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ are interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ are interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?lahef_rook

Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call clahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

```
call zlahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- mkl.fi

Description

The routine `?lahef_rook` computes a partial factorization of a complex Hermitian matrix A , using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

If $uplo = 'U'$:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^H & U_{22}^H \end{bmatrix}$$

If $uplo = 'L'$:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^H & L_{21}^H \\ 0 & I \end{bmatrix}$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

Note that U^H denotes the conjugate transpose of U .

This is an auxiliary routine called by `?hetrf_rook`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored:</p> <p>= 'U': upper triangular</p> <p>= 'L': lower triangular</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>. $n \geq 0$.</p>
<i>nb</i>	<p>INTEGER. The maximum number of columns of the matrix <i>A</i> that should be factored. <i>nb</i> should be at least 2 to allow for 2-by-2 pivot blocks.</p>
<i>a</i>	<p>COMPLEX for <code>clahef_rook</code></p> <p>DOUBLE COMPLEX for <code>zlahef_rook</code>.</p> <p>Array, DIMENSION (<i>lda</i>, <i>n</i>).</p> <p>On entry, the Hermitian matrix <i>A</i>.</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>A</i> contains the upper triangular part of the matrix <i>A</i>, and the strictly lower triangular part of <i>A</i> is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>A</i> contains the lower triangular part of the matrix <i>A</i>, and the strictly upper triangular part of <i>A</i> is not referenced.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. $lda \geq \max(1, n)$.</p>
<i>w</i>	<p>COMPLEX for <code>clahef_rook</code></p> <p>DOUBLE COMPLEX for <code>zlahef_rook</code>.</p> <p>Workspace array, DIMENSION (<i>ldw</i>, <i>nb</i>).</p>
<i>ldw</i>	<p>INTEGER. The leading dimension of the array <i>w</i>. $ldw \geq \max(1, n)$.</p>

Output Parameters

<i>kb</i>	<p>INTEGER. The number of columns of <i>A</i> that were actually factored <i>kb</i> is either <i>nb</i>-1 or <i>nb</i>, or <i>n</i> if $n \leq nb$.</p>
<i>a</i>	<p>On exit, <i>A</i> contains details of the partial factorization.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>). Details of the interchanges and the block structure of <i>D</i>.</p> <p>If <i>uplo</i> = 'U', only the last <i>kb</i> elements of <i>ipiv</i> are set;</p> <p>if <i>uplo</i> = 'L', only the first <i>kb</i> elements are set.</p> <p>If <i>ipiv</i>(<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i>(<i>k</i>) are interchanged and <i>D</i>(<i>k</i>, <i>k</i>) is a 1-by-1 diagonal block.</p>

If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k-1) < 0$, then rows and columns k and $-ipiv(k)$ are interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ are interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ are interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ are interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

call slatbs(*uplo*, *trans*, *diag*, *normin*, *n*, *kd*, *ab*, *ldab*, *x*, *scale*, *cnorm*, *info*)

call dlatbs(*uplo*, *trans*, *diag*, *normin*, *n*, *kd*, *ab*, *ldab*, *x*, *scale*, *cnorm*, *info*)

call clatbs(*uplo*, *trans*, *diag*, *normin*, *n*, *kd*, *ab*, *ldab*, *x*, *scale*, *cnorm*, *info*)

call zlatbs(*uplo*, *trans*, *diag*, *normin*, *n*, *kd*, *ab*, *ldab*, *x*, *scale*, *cnorm*, *info*)

Include Files

- mkl.fi

Description

The routine solves one of the triangular systems

$A*x = s*b$, or $A^T*x = s*b$, or $A^H*x = s*b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular band matrix. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?tbsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the matrix A is upper or lower triangular.

= 'U': upper triangular

= 'L': lower triangular

trans

CHARACTER*1.

Specifies the operation applied to A .

= 'N': solve $A*x = s*b$ (no transpose)

= 'T': solve $A^T*x = s*b$ (transpose)

	= 'C': solve $A^H*x = s*b$ (conjugate transpose)
<i>diag</i>	<p>CHARACTER*1.</p> <p>Specifies whether the matrix <i>A</i> is unit triangular</p> <p>= 'N': non-unit triangular</p> <p>= 'U': unit triangular</p>
<i>normin</i>	<p>CHARACTER*1.</p> <p>Specifies whether <i>cnorm</i> is set.</p> <p>= 'Y': <i>cnorm</i> contains the column norms on entry;</p> <p>= 'N': <i>cnorm</i> is not set on entry. On exit, the norms is computed and stored in <i>cnorm</i>.</p>
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>kd</i>	INTEGER. The number of subdiagonals or superdiagonals in the triangular matrix <i>A</i> . $kb \geq 0$.
<i>ab</i>	<p>REAL for slatbs</p> <p>DOUBLE PRECISION for dlatbs</p> <p>COMPLEX for clatbs</p> <p>DOUBLE COMPLEX for zlatbs.</p> <p>Array, DIMENSION (<i>ldab</i>, <i>n</i>).</p> <p>The upper or lower triangular band matrix <i>A</i>, stored in the first <i>kb</i>+1 rows of the array. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows:</p> <p>if <i>uplo</i> = 'U', $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$;</p> <p>if <i>uplo</i> = 'L', $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . $ldab \geq kb+1$.
<i>x</i>	<p>REAL for slatbs</p> <p>DOUBLE PRECISION for dlatbs</p> <p>COMPLEX for clatbs</p> <p>DOUBLE COMPLEX for zlatbs.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>On entry, the right hand side <i>b</i> of the triangular system.</p>
<i>cnorm</i>	<p>REAL for slatbs/clatbs</p> <p>DOUBLE PRECISION for dlatbs/zlatbs.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>If <i>NORMIN</i> = 'Y', <i>cnorm</i> is an input argument and <i>cnorm</i>(<i>j</i>) contains the norm of the off-diagonal part of the <i>j</i>-th column of <i>A</i>.</p>

If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or $'C'$, $cnorm(j)$ must be greater than or equal to the 1-norm.

Output Parameters

<i>scale</i>	REAL for slatbs/clatbs DOUBLE PRECISION for dlatbs/zlatbs. The scaling factor s for the triangular system as described above. If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $Ax = 0$.
<i>cnorm</i>	If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j -th column of A .
<i>info</i>	INTEGER. = 0: successful exit < 0: if $info = -k$, the k -th argument had an illegal value

?latm1

Computes the entries of a matrix as specified.

Syntax

```
call slatm1( mode, cond, irsign, idist, iseed, d, n, info )
call dlatm1( mode, cond, irsign, idist, iseed, d, n, info )
call clatm1( mode, cond, irsign, idist, iseed, d, n, info )
call zlatm1( mode, cond, irsign, idist, iseed, d, n, info )
```

Include Files

- mkl.fi

Description

The ?latm1 routine computes the entries of $D(1..n)$ as specified by *mode*, *cond* and *irsign*. *idist* and *iseed* determine the generation of random numbers.

?latm1 is called by slatmr (for slatm1 and dlatm1), and by clatmr (for clatm1 and zlatm1) to generate random test matrices for LAPACK programs.

Input Parameters

<i>mode</i>	INTEGER. On entry describes how d is to be computed: $mode = 0$ means do not change d . $mode = 1$ sets $d(1) = 1$ and $d(2:n) = 1.0/cond$ $mode = 2$ sets $d(1:n-1) = 1$ and $d(n) = 1.0/cond$ $mode = 3$ sets $d(i) = cond^{-(i-1)/(n-1)}$ $mode = 4$ sets $d(i) = 1 - (i-1)/(n-1) * (1 - 1/cond)$
-------------	---

$mode = 5$ sets d to random numbers in the range $(1/cond , 1)$ such that their logarithms are uniformly distributed.

$mode = 6$ sets d to random numbers from same distribution as the rest of the matrix.

$mode < 0$ has the same meaning as $abs(mode)$, except that the order of the elements of d is reversed.

Thus if $mode$ is positive, d has entries ranging from 1 to $1/cond$, if negative, from $1/cond$ to 1.

cond

REAL for slatm1,
 DOUBLE PRECISION for dlatm1,
 REAL for clatm1,
 DOUBLE PRECISION for zlatm1,

On entry, used as described under $mode$ above. If used, it must be ≥ 1 .

irsign

INTEGER.

On entry, if $mode$ is not -6, 0, or 6, determines sign of entries of d .

If $irsign = 0$, entries of d are unchanged.

If $irsign = 1$, each entry of d is multiplied by a random complex number uniformly distributed with absolute value 1.

idist

INTEGER. Specifies the distribution of the random numbers.

For slatm1 and dlatm1:

= 1: uniform (0,1)

= 2: uniform (-1,1)

= 3: normal (0,1)

For clatm1 and zlatm1:

= 1: real and imaginary parts each uniform (0,1)

= 2: real and imaginary parts each uniform (-1,1)

= 3: real and imaginary parts each normal (0,1)

= 4: complex number uniform in disk(0, 1)

iseed

INTEGER. Array, size (4).

Specifies the seed of the random number generator. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of $iseed(4)$ are changed on exit, and can be used in the next call to ?latm1 to continue the same random number sequence.

d

REAL for slatm1,
 DOUBLE PRECISION for dlatm1,
 COMPLEX for clatm1,
 DOUBLE COMPLEX for zlatm1,

	Array, size n .
n	INTEGER. Number of entries of d .
Output Parameters	
$iseed$	On exit, the seed is updated.
d	On exit, d is updated, unless $mode = 0$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -1$, $mode$ is not in range -6 to 6. If $info = -2$, $mode$ is neither -6, 0 nor 6, and $irsign$ is neither 0 nor 1. If $info = -3$, $mode$ is neither -6, 0 nor 6 and $cond$ is less than 1. If $info = -4$, $mode$ equals 6 or -6 and $idist$ is not in range 1 to 4. If $info = -7$, n is negative.

?latm2

Returns an entry of a random matrix.

Syntax

```
res = slatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork,
sparse )
```

```
res = dlatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork,
sparse )
```

```
res = clatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork,
sparse )
```

```
res = zlatm2( m, n, i, j, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork,
sparse )
```

Include Files

- mkl.fi

Description

The ?latm2 routine returns entry (i, j) of a random matrix of dimension (m, n) . It is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm2 differs from ?latm3 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm2, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3 can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers).

The matrix whose (i, j) entry is returned is constructed as follows (this routine only computes one entry):

- If i is outside $(1..m)$ or j is outside $(1..n)$, returns zero (this is convenient for generating matrices in band format).
- Generate a matrix A with random entries of distribution $idist$.
- Set the diagonal to D .
- Grade the matrix, if desired, from the left (by dl) and/or from the right (by dr or dl) as specified by $igrade$.
- Permute, if desired, the rows and/or columns as specified by $ipvtng$ and $iwork$.
- Band the matrix to have lower bandwidth kl and upper bandwidth ku .
- Set random entries to zero as specified by $sparse$.

Input Parameters

m	INTEGER. Number of rows of the matrix.
n	INTEGER. Number of columns of the matrix.
i	INTEGER. Row of the entry to be returned.
j	INTEGER. Column of the entry to be returned.
kl	INTEGER. Lower bandwidth.
ku	INTEGER. Upper bandwidth.
$idist$	<p>INTEGER. On entry, $idist$ specifies the type of distribution to be used to generate a random matrix .</p> <p>for <code>slatm2</code> and <code>dlatm2</code>:</p> <p>= 1: uniform (0,1)</p> <p>= 2: uniform (-1,1)</p> <p>= 3: normal (0,1)</p> <p>for <code>clatm2</code> and <code>zlatm2</code>:</p> <p>= 1: real and imaginary parts each uniform (0,1)</p> <p>= 2: real and imaginary parts each uniform (-1,1)</p> <p>= 3: real and imaginary parts each normal (0,1)</p> <p>= 4: complex number uniform in disk (0, 1)</p>
$iseed$	<p>INTEGER. Array, size 4.</p> <p>Seed for the random number generator.</p>
d	<p>REAL for <code>slatm2</code>,</p> <p>DOUBLE PRECISION for <code>dlatm2</code>,</p> <p>COMPLEX for <code>clatm2</code>,</p> <p>DOUBLE COMPLEX for <code>zlatm2</code>,</p> <p>Array, size $(\min(i, j))$. Diagonal entries of matrix.</p>
$igrade$	<p>INTEGER. Specifies grading of matrix as follows:</p> <p>= 0: no grading</p> <p>= 1: matrix premultiplied by <code>diag(dl)</code></p>

= 2: matrix postmultiplied by $\text{diag}(dr)$
 = 3: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dr)$
 = 4: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{inv}(\text{diag}(dl))$
 For *slatm2* and *slatm2*:
 = 5: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$
 For *clatm2* and *zlatm2*:
 = 5: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(\text{conjg}(dl))$
 = 6: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$

dl REAL for *slatm2*,
 DOUBLE PRECISION for *dlatm2*,
 COMPLEX for *clatm2*,
 DOUBLE COMPLEX for *zlatm2*,
 Array, size (*i* or *j*), as appropriate.
 Left scale factors for grading matrix.

dr REAL for *slatm2*,
 DOUBLE PRECISION for *dlatm2*,
 COMPLEX for *clatm2*,
 DOUBLE COMPLEX for *zlatm2*,
 Array, size (*i* or *j*), as appropriate.
 Right scale factors for grading matrix.

ipvtng INTEGER. On entry specifies pivoting permutations as follows:
 = 0: none
 = 1: row pivoting
 = 2: column pivoting
 = 3: full pivoting, i.e., on both sides

iwork INTEGER.
 Array, size (*i* or *j*), as appropriate. This array specifies the permutation used. The row (or column) in position *k* was originally in position *iwork*(*k*). This differs from *iwork* for *?latm3*.

sparse REAL for *slatm2*,
 DOUBLE PRECISION for *dlatm2*,
 REAL for *clatm2*,
 DOUBLE PRECISION for *zlatm2*,

Specifies the sparsity of the matrix. If sparse matrix is to be generated, *sparse* should lie between 0 and 1. A uniform (0, 1) random number *x* is generated and compared to *sparse*. If *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.

Output Parameters

<i>iseed</i>	INTEGER. On exit, the seed is updated.
<i>res</i>	REAL for slatm2, DOUBLE PRECISION for dlatm2, COMPLEX for clatm2, DOUBLE COMPLEX for zlatm2, Entry of a random matrix.

?latm3

Returns set entry of a random matrix.

Syntax

```
res = slatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

```
res = dlatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

```
res = clatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

```
res = zlatm3( m, n, i, j, isub, jsub, kl, ku, idist, iseed, d, igrade, dl, dr, ipvtng, iwork, sparse )
```

Include Files

- mkl.fi

Description

The ?latm3 routine returns the (*isub*, *jsub*) entry of a random matrix of dimension (*m*, *n*) described by the other parameters. (*isub*, *jsub*) is the final position of the (*i*, *j*) entry after pivoting according to *ipvtng* and *iwork*. ?latm3 is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm3 differs from ?latm2 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm3, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3 can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers in different orders for different pivot orders).

The matrix whose (*isub*, *jsub*) entry is returned is constructed as follows (this routine only computes one entry):

- If *isub* is outside (1..*m*) or *jsub* is outside (1..*n*), returns zero (this is convenient for generating matrices in band format).
- Generate a matrix *A* with random entries of distribution *idist*.
- Set the diagonal to *D*.
- Grade the matrix, if desired, from the left (by *dl*) and/or from the right (by *dr* or *dl*) as specified by *igrade*.
- Permute, if desired, the rows and/or columns as specified by *ipvtng* and *iwork*.
- Band the matrix to have lower bandwidth *kl* and upper bandwidth *ku*.
- Set random entries to zero as specified by *sparse*.

Input Parameters

<i>m</i>	INTEGER. Number of rows of matrix.
<i>n</i>	INTEGER. Number of columns of matrix.
<i>i</i>	INTEGER. Row of unpivoted entry to be returned.
<i>j</i>	INTEGER. Column of unpivoted entry to be returned.
<i>isub</i>	INTEGER. Row of pivoted entry to be returned.
<i>jsub</i>	INTEGER. Column of pivoted entry to be returned.
<i>kl</i>	INTEGER. Lower bandwidth.
<i>ku</i>	INTEGER. Upper bandwidth.
<i>idist</i>	<p>INTEGER. On entry, <i>idist</i> specifies the type of distribution to be used to generate a random matrix.</p> <p>for <i>slatm2</i> and <i>dlatm2</i>:</p> <p>= 1: uniform (0,1)</p> <p>= 2: uniform (-1,1)</p> <p>= 3: normal (0,1)</p> <p>for <i>clatm2</i> and <i>zlatm2</i>:</p> <p>= 1: real and imaginary parts each uniform (0,1)</p> <p>= 2: real and imaginary parts each uniform (-1,1)</p> <p>= 3: real and imaginary parts each normal (0,1)</p> <p>= 4: complex number uniform in disk(0, 1)</p>
<i>iseed</i>	<p>INTEGER. Array, size 4.</p> <p>Seed for random number generator.</p>
<i>d</i>	<p>REAL for <i>slatm3</i>,</p> <p>DOUBLE PRECISION for <i>dlatm3</i>,</p> <p>COMPLEX for <i>clatm3</i>,</p> <p>DOUBLE COMPLEX for <i>zlatm3</i>,</p> <p>Array, size (min(<i>i</i>, <i>j</i>)). Diagonal entries of matrix.</p>

igrade INTEGER. Specifies grading of matrix as follows:

- = 0: no grading
- = 1: matrix premultiplied by $\text{diag}(dl)$
- = 2: matrix postmultiplied by $\text{diag}(dr)$
- = 3: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dr)$
- = 4: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{inv}(\text{diag}(dl))$

For *slatm2* and *slatm2*:

- = 5: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$

For *clatm2* and *zlatm2*:

- = 5: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(\text{conj}(dl))$
- = 6: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$

dl REAL for *slatm3*,
DOUBLE PRECISION for *dlatm3*,
COMPLEX for *clatm3*,
DOUBLE COMPLEX for *zlatm3*,
Array, size (*i* or *j*, as appropriate).
Left scale factors for grading matrix.

dr REAL for *slatm3*,
DOUBLE PRECISION for *dlatm3*,
COMPLEX for *clatm3*,
DOUBLE COMPLEX for *zlatm3*,
Array, size (*i* or *j*, as appropriate).
Right scale factors for grading matrix.

ipvtng INTEGER. On entry specifies pivoting permutations as follows:

- If *ipvtng* = 0: none.
- If *ipvtng* = 1: row pivoting.
- If *ipvtng* = 2: column pivoting.
- If *ipvtng* = 3: full pivoting, i.e., on both sides.

sparse REAL for *slatm3*,
DOUBLE PRECISION for *dlatm3*,
REAL for *clatm3*,
DOUBLE PRECISION for *zlatm3*,

On entry, specifies the sparsity of the matrix if sparse matrix is to be generated. *sparse* should lie between 0 and 1. A uniform(0, 1) random number *x* is generated and compared to *sparse*; if *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.

iwork

INTEGER.

Array, size (*i* or *j*, as appropriate). This array specifies the permutation used. The row (or column) originally in position *k* is in position *iwork*(*k*) after pivoting. This differs from *iwork* for ?latm2.

Output Parameters

isub

On exit, row of pivoted entry is updated.

jsub

On exit, column of pivoted entry is updated.

iseed

On exit, the seed is updated.

res

REAL for slatm3,

DOUBLE PRECISION for dlatm3,

COMPLEX for clatm3,

DOUBLE COMPLEX for zlatm3,

Entry of a random matrix.

?latm5

Generates matrices involved in the Generalized Sylvester equation.

Syntax

```
call slatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
            ldl, alpha, qblcka, qblckb )
```

```
call dlatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
            ldl, alpha, qblcka, qblckb )
```

```
call clatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
            ldl, alpha, qblcka, qblckb )
```

```
call zlatm5( prtype, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, r, ldr, l,
            ldl, alpha, qblcka, qblckb )
```

Include Files

- mkl.fi

Description

The ?latm5 routine generates matrices involved in the Generalized Sylvester equation:

$$A * R - L * B = C$$

$$D * R - L * E = F$$

They also satisfy the diagonalization condition:

$$\begin{bmatrix} I & -L \\ & I \end{bmatrix} \begin{bmatrix} A & -C \\ & B \end{bmatrix} \begin{bmatrix} I & R \\ & I \end{bmatrix} = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

$$\begin{bmatrix} I & -L \\ & I \end{bmatrix} \begin{bmatrix} D & -F \\ & E \end{bmatrix} \begin{bmatrix} I & R \\ & I \end{bmatrix} = \begin{bmatrix} D & \\ & E \end{bmatrix}$$

Input Parameters

prtype

INTEGER. Specifies the type of matrices to generate.

- If *prtype* = 1, *A* and *B* are Jordan blocks, *D* and *E* are identity matrices.

A:

If (*i* == *j*) then $A_{i,j} = 1.0$.

If (*j* == *i* + 1) then $A_{i,j} = -1.0$.

Otherwise $A_{i,j} = 0.0$, *i*, *j* = 1...*m*

B:

If (*i* == *j*) then $B_{i,j} = 1.0 - \text{alpha}$.

If (*j* == *i* + 1) then $B_{i,j} = 1.0$.

Otherwise $B_{i,j} = 0.0$, *i*, *j* = 1...*n*.

D:

If (*i* == *j*) then $D_{i,j} = 1.0$.

Otherwise $D_{i,j} = 0.0$, *i*, *j* = 1...*m*.

E:

If (*i* == *j*) then $E_{i,j} = 1.0$

Otherwise $E_{i,j} = 0.0$, *i*, *j* = 1...*n*.

L = *R* are chosen from [-10...10], which specifies the right hand sides (*C*, *F*).

- If *prtype* = 2 or 3: Triangular and/or quasi- triangular.

A:

If (*i* ≤ *j*) then $A_{i,j} = [-1...1]$.

Otherwise $A_{i,j} = 0.0$, *i*, *j* = 1...*M*.

If (*prtype* = 3) then $A_{k+1,k+1} = A_{k,k}$;

$A_{k+1,k} = [-1...1]$;

$\text{sign}(A_{k,k+1}) = -(\text{sign}(A_{k+1,k}))$.

k = 1, *m*- 1, *qblkca*

B:

If (*i* ≤ *j*) then $B_{i,j} = [-1...1]$.

Otherwise $B_{i,j} = 0.0$, *i*, *j* = 1...*n*.

If (*prtype* = 3) then $B_{k+1,k+1} = B_{k,k}$

$B_{k+1,k} = [-1...1]$

$$\text{sign}(B_{k,k+1}) = -(\text{sign}(B_{k+1,k}))$$

$$k = 1, n - 1, \text{qblckb.}$$

D:

$$\text{If } (i \leq j) \text{ then } D_{i,j} = [-1 \dots 1].$$

$$\text{Otherwise } D_{i,j} = 0.0, \quad i, j = 1 \dots m.$$

E:

$$\text{If } (i \leq j) \text{ then } E_{i,j} = [-1 \dots 1].$$

$$\text{Otherwise } E_{i,j} = 0.0, \quad i, j = 1 \dots N.$$

L, R are chosen from $[-10 \dots 10]$, which specifies the right hand sides (*C, F*).

- If *prtype* = 4 Full

$$A_{i,j} = [-10 \dots 10]$$

$$D_{i,j} = [-1 \dots 1] \quad i, j = 1 \dots m$$

$$B_{i,j} = [-10 \dots 10]$$

$$E_{i,j} = [-1 \dots 1] \quad i, j = 1 \dots n$$

$$R_{i,j} = [-10 \dots 10]$$

$$L_{i,j} = [-1 \dots 1] \quad i = 1 \dots m, j = 1 \dots n$$

L and *R* specifies the right hand sides (*C, F*).

- If *prtype* = 5 special case common and/or close eigs.

<i>m</i>	INTEGER. Specifies the order of <i>A</i> and <i>D</i> and the number of rows in <i>C, F, R</i> and <i>L</i> .
<i>n</i>	INTEGER. Specifies the order of <i>B</i> and <i>E</i> and the number of columns in <i>C, F, R</i> and <i>L</i> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> .
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> .
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> .
<i>ldd</i>	INTEGER. The leading dimension of <i>d</i> .
<i>lde</i>	INTEGER. The leading dimension of <i>e</i> .
<i>ldf</i>	INTEGER. The leading dimension of <i>f</i> .
<i>ldr</i>	INTEGER. The leading dimension of <i>r</i> .
<i>ldl</i>	INTEGER. The leading dimension of <i>l</i> .
<i>alpha</i>	REAL for <i>slatm5</i> , DOUBLE PRECISION for <i>dlatm5</i> , REAL for <i>clatm5</i> , DOUBLE PRECISION for <i>zlatm5</i> ,

Parameter used in generating $prtype = 1$ and 5 matrices.

- qblcka* INTEGER. When $prtype = 3$, specifies the distance between 2-by-2 blocks on the diagonal in *A*. Otherwise, *qblcka* is not referenced. $qblcka > 1$.
- qblckb* INTEGER. When $prtype = 3$, specifies the distance between 2-by-2 blocks on the diagonal in *B*. Otherwise, *qblckb* is not referenced. $qblckb > 1$.

Output Parameters

- a* REAL for *slatm5*,
 DOUBLE PRECISION for *dlatm5*,
 COMPLEX for *clatm5*,
 DOUBLE COMPLEX for *zlatm5*,
 Array, size (*lda*, *m*). On exit *a* contains *m*-by-*m* array *A* initialized according to $prtype$.
- b* REAL for *slatm5*,
 DOUBLE PRECISION for *dlatm5*,
 COMPLEX for *clatm5*,
 DOUBLE COMPLEX for *zlatm5*,
 Array, size (*ldb*, *n*). On exit *b* contains the *n*-by-*n* array *B* initialized according to $prtype$.
- c* REAL for *slatm5*,
 DOUBLE PRECISION for *dlatm5*,
 COMPLEX for *clatm5*,
 DOUBLE COMPLEX for *zlatm5*,
 Array, size (*ldc*, *n*). On exit *c* contains the *m*-by-*n* array *C* initialized according to $prtype$.
- d* REAL for *slatm5*,
 DOUBLE PRECISION for *dlatm5*,
 COMPLEX for *clatm5*,
 DOUBLE COMPLEX for *zlatm5*,
 Array, size (*ldd*, *m*). On exit *d* contains the *m*-by-*m* array *D* initialized according to $prtype$.
- e* REAL for *slatm5*,
 DOUBLE PRECISION for *dlatm5*,
 COMPLEX for *clatm5*,
 DOUBLE COMPLEX for *zlatm5*,
 Array, size (*lde*, *n*). On exit *e* contains the *n*-by-*n* array *E* initialized according to $prtype$.

<i>f</i>	<p>REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (<i>ldf</i>, <i>n</i>). On exit <i>f</i> contains the <i>m</i>-by-<i>n</i> array <i>F</i> initialized according to <i>prtype</i>.</p>
<i>r</i>	<p>REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (<i>ldr</i>, <i>n</i>). On exit <i>R</i> contains the <i>m</i>-by-<i>n</i> array <i>R</i> initialized according to <i>prtype</i>.</p>
<i>l</i>	<p>REAL for slatm5, DOUBLE PRECISION for dlatm5, COMPLEX for clatm5, DOUBLE COMPLEX for zlatm5, Array, size (<i>ldl</i>, <i>n</i>). On exit <i>l</i> contains the <i>m</i>-by-<i>n</i> array <i>L</i> initialized according to <i>prtype</i>.</p>

?latm6

Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

Syntax

```
call slatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call dlatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call clatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
call zlatm6( type, n, a, lda, b, x, ldx, y, ldy, alpha, beta, wx, wy, s, dif )
```

Include Files

- mkl.fi

Description

The ?latm6 routine generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

There two kinds of test matrix pairs:

$$(A, B) = \text{inverse}(YH) * (Da, Db) * \text{inverse}(X)$$

Type 1:

$$Da = \begin{bmatrix} 1+a & 0 & 0 & 0 & 0 \\ 0 & 2+a & 0 & 0 & 0 \\ 0 & 0 & 3+a & 0 & 0 \\ 0 & 0 & 0 & 4+a & 0 \\ 0 & 0 & 0 & 0 & 5+2 \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Type 2:

$$Da = \begin{bmatrix} 1+i & 0 & 0 & 0 & 0 \\ 0 & 1-i & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & (1+a)+(1+b)i & 0 \\ 0 & 0 & 0 & 0 & (1+a)-(1+b)i \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In both cases the same inverse(YH) and inverse(X) are used to compute (A, B), giving the exact eigenvectors to (A,B) as (YH, X):

$$YH = \begin{bmatrix} 1 & 0 & -y & y & -y \\ 0 & 1 & -y & y & -y \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 0 & -x & -x & x \\ 0 & 1 & x & -x & -x \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

where a, b, x and y will have all values independently of each other.

Input Parameters

<i>type</i>	INTEGER. Specifies the problem type.
<i>n</i>	INTEGER. Size of the matrices <i>A</i> and <i>B</i> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> and of <i>b</i> .
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> .
<i>ldy</i>	INTEGER. The leading dimension of <i>y</i> .
<i>alpha, beta</i>	REAL for slatm6, DOUBLE PRECISION for dlatm6, COMPLEX for clatm6, DOUBLE COMPLEX for zlatm6, Weighting constants for matrix <i>A</i> .
<i>wx</i>	REAL for slatm6, DOUBLE PRECISION for dlatm6, COMPLEX for clatm6,

DOUBLE COMPLEX for `zlatm6`,
 Constant for right eigenvector matrix.

`wy` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 COMPLEX for `clatm6`,
 DOUBLE COMPLEX for `zlatm6`,
 Constant for left eigenvector matrix.

Output Parameters

`a` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 COMPLEX for `clatm6`,
 DOUBLE COMPLEX for `zlatm6`,
 Array, size (lda, n) . On exit, `a` contains the n -by- n matrix initialized according to `type`.

`b` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 COMPLEX for `clatm6`,
 DOUBLE COMPLEX for `zlatm6`,
 Array, size (lda, n) . On exit, `b` contains the n -by- n matrix initialized according to `type`.

`x` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 COMPLEX for `clatm6`,
 DOUBLE COMPLEX for `zlatm6`,
 Array, size (ldx, n) . On exit, `x` contains the n -by- n matrix of right eigenvectors.

`y` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 COMPLEX for `clatm6`,
 DOUBLE COMPLEX for `zlatm6`,
 Array, size (ldy, n) . On exit, `y` is the n -by- n matrix of left eigenvectors.

`s` REAL for `slatm6`,
 DOUBLE PRECISION for `dlatm6`,
 REAL for `clatm6`,
 DOUBLE PRECISION for `zlatm6`,

Array, size (n). $s(i)$ is the reciprocal condition number for eigenvalue i .

dif

REAL for slatm6,

DOUBLE PRECISION for dlatm6,

REAL for clatm6,

DOUBLE PRECISION for zlatm6,

Array, size(n). $dif(i)$ is the reciprocal condition number for eigenvector i .

?latme

Generates random non-symmetric square matrices with specified eigenvalues.

Syntax

```
call slatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes,
conds, kl, ku, anorm, a, lda, work, info )
```

```
call dlatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes,
conds, kl, ku, anorm, a, lda, work, info )
```

```
call clatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes,
conds, kl, ku, anorm, a, lda, work, info )
```

```
call zlatme( n, dist, iseed, d, mode, cond, dmax, ei, rsign, upper, sim, ds, modes,
conds, kl, ku, anorm, a, lda, work, info )
```

Include Files

- mkl.fi

Description

The ?latme routine generates random non-symmetric square matrices with specified eigenvalues. ?latme operates by applying the following sequence of operations:

1. Set the diagonal to d , where d may be input or computed according to $mode$, $cond$, $dmax$, and $rsign$ as described below.
2. If $upper = 'T'$, the upper triangle of a is set to random values out of distribution $dist$.
3. If $sim='T'$, a is multiplied on the left by a random matrix X , whose singular values are specified by ds , $modes$, and $conds$, and on the right by X inverse.
4. If $kl < n-1$, the lower bandwidth is reduced to kl using Householder transformations. If $ku < n-1$, the upper bandwidth is reduced to ku .
5. If $anorm$ is not negative, the matrix is scaled to have maximum-element-norm $anorm$.

NOTE

Since the matrix cannot be reduced beyond Hessenberg form, no packing options are available.

Input Parameters

n

INTEGER. The number of columns (or rows) of A .

dist CHARACTER*1. On entry, *dist* specifies the type of distribution to be used to generate the random eigen-/singular values, and on the upper triangle (see *upper*).

If *dist* = 'U': uniform(0, 1)

If *dist* = 'S': uniform(-1, 1)

If *dist* = 'N': normal(0, 1)

If *dist* = 'D': uniform on the complex disc $|z| < 1$.

iseed INTEGER. Array, size 4.

On entry *iseed* specifies the seed of the random number generator. The elements should lie between 0 and 4095 inclusive, and *iseed*(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.

d REAL for *slatme*,
DOUBLE PRECISION for *dlatme*,
COMPLEX for *clatme*,
DOUBLE COMPLEX for *zlatme*,
Array, size (*n*). This array is used to specify the eigenvalues of *A*.
If *mode* = 0, then *d* is assumed to contain the eigenvalues. Otherwise they are computed according to *mode*, *cond*, *dmax*, and *rsign* and placed in *d*.

mode INTEGER. On entry *mode* describes how the eigenvalues are to be specified:

mode = 0 means use *d* (with *ei* for *slatme* and *dlatme*) as input.

mode = 1 sets $d(1) = 1$ and $d[1:n - 1] = 1.0/cond$.

mode = 2 sets $d(1:n-1) = 1$ and $d(n) = 1.0/cond$.

mode = 3 sets $d(i) = cond^{-(i-1)/(n-1)}$.

mode = 4 sets $d(i) = 1 - (i-1)/(n-1) * (1 - 1/cond)$.

mode = 5 sets *d* to random numbers in the range ($1/cond$, 1) such that their logarithms are uniformly distributed.

mode = 6 sets *d* to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as $abs(mode)$, except that the order of the elements of *d* is reversed.

Thus if *mode* is between 1 and 4, *d* has entries ranging from 1 to $1/cond$, if between -1 and -4, *d* has entries ranging from $1/cond$ to 1.

cond REAL for *slatme*,
DOUBLE PRECISION for *dlatme*,
REAL for *clatme*,
DOUBLE PRECISION for *zlatme*,

On entry, this is used as described under *mode* above. If used, it must be ≥ 1 .

dmax

REAL for *slatme*,
 DOUBLE PRECISION for *dlatme*,
 COMPLEX for *clatme*,
 DOUBLE COMPLEX for *zlatme*,

If *mode* is not -6, 0 or 6, the contents of *d* as computed according to *mode* and *cond* are scaled by $dmax / \max(\text{abs}(d(i)))$. Note that *dmax* needs not be positive or real: if *dmax* is negative or complex (or zero), *d* will be scaled by a negative or complex number (or zero). If *rsign*='F' then the largest (absolute) eigenvalue will be equal to *dmax*.

ei

CHARACTER*1. Used by *slatme* and *dlatme* only.

Array, size (*n*).

If *mode* = 0, and *ei*(1) is not ' ' (space character), this array specifies which elements of *d* (on input) are real eigenvalues and which are the real and imaginary parts of a complex conjugate pair of eigenvalues. The elements of *ei* may then only have the values 'R' and 'I'.

If *ei*(*j*) = 'R' and *ei*(*j* + 1) = 'I', then the *j*-th eigenvalue is $\text{cmplx}(d(j), d(j + 1))$, and the (*j* + 1)-th is the complex conjugate thereof.

If *ei*(*j*) = *ei*(*j* + 1)='R', then the *j*-th eigenvalue is *d*(*j*) (i.e., real). *ei*(1) may not be 'I', nor may two adjacent elements of *ei* both have the value 'I'.

If *mode* is not 0, then *ei* is ignored. If *mode* is 0 and *ei*(1) = ' ', then the eigenvalues will all be real.

rsign

CHARACTER*1. If *mode* is not 0, 6, or -6, and *rsign* = 'T', then the elements of *d*, as computed according to *mode* and *cond*, are multiplied by a random sign (+1 or -1) for *slatme* and *dlatme* or by a complex number from the unit circle $|z| = 1$ for *clatme* and *zlatme*.

If *rsign* = 'F', the elements of *d* are not multiplied. *rsign* may only have the values 'T' or 'F'.

upper

CHARACTER*1. If *upper* = 'T', then the elements of *a* above the diagonal will be set to random numbers out of *dist*.

If *upper* = 'F', they will not. *upper* may only have the values 'T' or 'F'.

sim

CHARACTER*1. If *sim* = 'T', then *a* will be operated on by a "similarity transform", i.e., multiplied on the left by a matrix *X* and on the right by *X* inverse. $X = USV$, where *U* and *V* are random unitary matrices and *S* is a (diagonal) matrix of singular values specified by *ds*, *modes*, and *conds*.

If *sim* = 'F', then *a* will not be transformed.

ds

REAL for *slatme*,
 DOUBLE PRECISION for *dlatme*,

REAL for `clatme`,

DOUBLE PRECISION for `zlatme`,

This array is used to specify the singular values of X , in the same way that d specifies the eigenvalues of a . If $mode = 0$, the ds contains the singular values, which may not be zero.

`modes` INTEGER.

Similar to `mode`, but for specifying the diagonal of S . `modes = -6` and `+6` are not allowed (since they would result in randomly ill-conditioned eigenvalues.)

`conds` REAL for `slatme`,
DOUBLE PRECISION for `dlatme`,
REAL for `clatme`,
DOUBLE PRECISION for `zlatme`,

Similar to `cond`, but for specifying the diagonal of S .

`kl` INTEGER. This specifies the lower bandwidth of the matrix. `kl = 1` specifies upper Hessenberg form. If `kl` is at least $n-1$, then A will have full lower bandwidth.

`ku` INTEGER. This specifies the upper bandwidth of the matrix. `ku = 1` specifies lower Hessenberg form.

If `ku` is at least $n-1$, then a will have full upper bandwidth.

If `ku` and `kl` are both at least $n-1$, then a will be dense. Only one of `ku` and `kl` may be less than $n-1$.

`anorm` REAL for `slatme`,
DOUBLE PRECISION for `dlatme`,
REAL for `clatme`,
DOUBLE PRECISION for `zlatme`,

If `anorm` is not negative, then a is scaled by a non-negative real number to make the maximum-element-norm of a to be `anorm`.

`lda` INTEGER. Number of rows of matrix A .

`work` REAL for `slatme`,
DOUBLE PRECISION for `dlatme`,
COMPLEX for `clatme`,
DOUBLE COMPLEX for `zlatme`,
Array, size $(3*n)$. Workspace.

Output Parameters

`iseed` INTEGER.

On exit, the seed is updated.

d Modified if *mode* is nonzero.

ds Modified if *mode* is nonzero.

a REAL for *slatme*,
 DOUBLE PRECISION for *dlatme*,
 COMPLEX for *clatme*,
 DOUBLE COMPLEX for *zlatme*,
 Array, size (*lda*, *n*). On exit, *a* is the desired test matrix.

info INTEGER.
 If *info* = 0, execution is successful.
 If *info* = -1, *n* is negative .
 If *info* = -2, *dist* is an illegal string.
 If *info* = -5, *mode* is not in range -6 to 6.
 If *info* = -6, *cond* is less than 1.0, and *mode* is not -6, 0, or 6 .
 If *info* = -9, *rsign* is not 'T' or 'F' .
 If *info* = -10, *upper* is not 'T' or 'F'.
 If *info* = -11, *sim* is not 'T' or 'F'.
 If *info* = -12, *modes* = 0 and *ds* has a zero singular value.
 If *info* = -13, *modes* is not in the range -5 to 5.
 If *info* = -14, *modes* is nonzero and *conds* is less than 1. .
 If *info* = -15, *kl* is less than 1.
 If *info* = -16, *ku* is less than 1, or *kl* and *ku* are both less than *n*-1.
 If *info* = -19, *lda* is less than *m*.
 If *info* = 1, error return from ?latm1 (computing *d*) .
 If *info* = 2, cannot scale to *dmax* (max. eigenvalue is 0) .
 If *info* = 3, error return from *slatm1*(for *slatme* and *clatme*), *dlatm1*
 (for *dlatme* and *zlatme*) .
 If *info* = 4, error return from ?large.
 If *info* = 5, zero singular value from *slatm1*(for *slatme* and *clatme*),
dlatm1(for *dlatme* and *zlatme*).

?latmr

Generates random matrices of various types.

Syntax

call *slatmr* (*m*, *n*, *dist*, *iseed*, *sym*, *d*, *mode*, *cond*, *dmax*, *rsign*, *grade*, *dl*, *model*,
concl, *dr*, *moder*, *condr*, *pivtng*, *ipivot*, *kl*, *ku*, *sparse*, *anorm*, *pack*, *a*, *lda*, *iwork*,
info)

```
call dlatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtnng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
```

```
call clatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtnng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
```

```
call zlatmr (m, n, dist, iseed, sym, d, mode, cond, dmax, rsign, grade, dl, model,
condl, dr, moder, condr, pivtnng, ipivot, kl, ku, sparse, anorm, pack, a, lda, iwork,
info)
```

Description

The `?latmr` routine operates by applying the following sequence of operations:

1. Generate a matrix A with random entries of distribution $dist$:
 - If $sym = 'S'$, the matrix is symmetric,
 - If $sym = 'H'$, the matrix is Hermitian,
 - If $sym = 'N'$, the matrix is nonsymmetric.
2. Set the diagonal to D , where D may be input or computed according to $mode$, $cond$, $dmax$ and $rsign$ as described below.
3. Grade the matrix, if desired, from the left or right as specified by $grade$. The inputs dl , $model$, $condl$, dr , $moder$ and $condr$ also determine the grading as described below.
4. Permute, if desired, the rows and/or columns as specified by $pivtnng$ and $ipivot$.
5. Set random entries to zero, if desired, to get a random sparse matrix as specified by $sparse$.
6. Make A a band matrix, if desired, by zeroing out the matrix outside a band of lower bandwidth kl and upper bandwidth ku .
7. Scale A , if desired, to have maximum entry $anorm$.
8. Pack the matrix if desired. See options specified by the $pack$ parameter.

NOTE

If two calls to `?latmr` differ only in the $pack$ parameter, they generate mathematically equivalent matrices. If two calls to `?latmr` both have full bandwidth ($kl = m-1$ and $ku = n-1$), and differ only in the $pivtnng$ and $pack$ parameters, then the matrices generated differ only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be and is not maintained with less than full bandwidth.

Input Parameters

m	INTEGER. Number of rows of A .
n	INTEGER. Number of columns of A .
$dist$	CHARACTER. On entry, $dist$ specifies the type of distribution to be used to generate a random matrix . <ul style="list-style-type: none"> If $dist = 'U'$, real and imaginary parts are independent uniform(0, 1). If $dist = 'S'$, real and imaginary parts are independent uniform(-1, 1). If $dist = 'N'$, real and imaginary parts are independent normal(0, 1). If $dist = 'D'$, distribution is uniform on interior of unit disk.

<i>iseed</i>	<p>INTEGER. Array, size 4.</p> <p>On entry, <i>iseed</i> specifies the seed of the random number generator. They should lie between 0 and 4095 inclusive, and <i>iseed</i>(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.</p>
<i>sym</i>	<p>CHARACTER. If <i>sym</i> = 'S', generated matrix is symmetric.</p> <p>If <i>sym</i> = 'H', generated matrix is Hermitian.</p> <p>If <i>sym</i> = 'N', generated matrix is nonsymmetric.</p>
<i>d</i>	<p>REAL for slatmr, DOUBLE PRECISION for dlatmr, COMPLEX for clatmr, DOUBLE COMPLEX for zlatmr,</p> <p>On entry this array specifies the diagonal entries of the diagonal of <i>A</i>. <i>d</i> may either be specified on entry, or set according to <i>mode</i> and <i>cond</i> as described below. If the matrix is Hermitian, the real part of <i>d</i> is taken. May be changed on exit if <i>mode</i> is nonzero.</p>
<i>mode</i>	<p>INTEGER. On entry describes how <i>d</i> is to be used:</p> <p><i>mode</i> = 0 means use <i>d</i> as input.</p> <p><i>mode</i> = 1 sets $d(1)=1$ and $d(2:n)=1.0/cond$.</p> <p><i>mode</i> = 2 sets $d(1:n-1)=1$ and $d(n)=1.0/cond$.</p> <p><i>mode</i> = 3 sets $d(i)=cond^{*(-(i-1)/(n-1))}$.</p> <p><i>mode</i> = 4 sets $d(i)=1 - (i-1)/(n-1)*(1 - 1/cond)$.</p> <p><i>mode</i> = 5 sets <i>d</i> to random numbers in the range ($1/cond$, 1) such that their logarithms are uniformly distributed.</p> <p><i>mode</i> = 6 sets <i>d</i> to random numbers from same distribution as the rest of the matrix.</p> <p><i>mode</i> < 0 has the same meaning as $abs(mode)$, except that the order of the elements of <i>d</i> is reversed.</p> <p>Thus if <i>mode</i> is between 1 and 4, <i>d</i> has entries ranging from 1 to $1/cond$, if between -1 and -4, <i>D</i> has entries ranging from $1/cond$ to 1.</p>
<i>cond</i>	<p>REAL for slatmr, DOUBLE PRECISION for dlatmr, REAL for clatmr, DOUBLE PRECISION for zlatmr,</p> <p>On entry, used as described under <i>mode</i> above. If used, <i>cond</i> must be ≥ 1.</p>
<i>dmax</i>	<p>REAL for slatmr, DOUBLE PRECISION for dlatmr,</p>

COMPLEX for `clatmr`,

DOUBLE COMPLEX for `zlatmr`,

If `mode` is not -6, 0, or 6, the diagonal is scaled by $dmax / \max(\text{abs}(d(i)))$, so that maximum absolute entry of diagonal is $\text{abs}(dmax)$. If `dmax` is complex (or zero), the diagonal is scaled by a complex number (or zero).

`rsign`

CHARACTER. If `mode` is not -6, 0, or 6, specifies the sign of the diagonal as follows:

For `slatmr` and `dlatmr`, if `rsign` = 'T', diagonal entries are multiplied 1 or -1 with a probability of 0.5.

For `clatmr` and `zlatmr`, if `rsign` = 'T', diagonal entries are multiplied by a random complex number uniformly distributed with absolute value 1.

If `rsign` = 'F', diagonal entries are unchanged.

`grade`

CHARACTER. Specifies grading of matrix as follows:

If `grade` = 'N', there is no grading

If `grade` = 'L', matrix is premultiplied by `diag(dl)` (only if matrix is nonsymmetric)

If `grade` = 'R', matrix is postmultiplied by `diag(dr)` (only if matrix is nonsymmetric)

If `grade` = 'B', matrix is premultiplied by `diag(dl)` and postmultiplied by `diag(dr)` (only if matrix is nonsymmetric)

If `grade` = 'H', matrix is premultiplied by `diag(dl)` and postmultiplied by `diag(conjg(dl))` (only if matrix is Hermitian or nonsymmetric)

If `grade` = 'S', matrix is premultiplied by `diag(dl)` and postmultiplied by `diag(dl)` (only if matrix is symmetric or nonsymmetric)

If `grade` = 'E', matrix is premultiplied by `diag(dl)` and postmultiplied by `inv(diag(dl))` (only if matrix is nonsymmetric)

NOTE

if `grade` = 'E', then `m` must equal `n`.

`dl`

REAL for `slatmr`,

DOUBLE PRECISION for `dlatmr`,

COMPLEX for `clatmr`,

DOUBLE COMPLEX for `zlatmr`,

Array, size (`m`).

If `model` = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under `grade` above.

If `model` is not zero, then `dl` is set according to `model` and `condl`, analogous to the way `D` is set according to `mode` and `cond` (except there is no `dmax` parameter for `dl`).

If `grade` = 'E', then `dl` cannot have zero entries.

	Not referenced if $grade = 'N'$ or $'R'$. Changed on exit.
<i>model</i>	INTEGER. This specifies how the diagonal array <i>dl</i> is computed, just as <i>mode</i> specifies how <i>D</i> is computed.
<i>condl</i>	REAL for <i>slatmr</i> , DOUBLE PRECISION for <i>dlatmr</i> , REAL for <i>clatmr</i> , DOUBLE PRECISION for <i>zlatmr</i> , When <i>model</i> is not zero, this specifies the condition number of the computed <i>dl</i> .
<i>dr</i>	REAL for <i>slatmr</i> , DOUBLE PRECISION for <i>dlatmr</i> , COMPLEX for <i>clatmr</i> , DOUBLE COMPLEX for <i>zlatmr</i> , If <i>moder</i> = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under <i>grade</i> above. If <i>moder</i> is not zero, then <i>dr</i> is set according to <i>moder</i> and <i>condr</i> , analogous to the way <i>d</i> is set according to <i>mode</i> and <i>cond</i> (except there is no <i>dmax</i> parameter for <i>dr</i>). Not referenced if $grade = 'N', 'L', 'H', 'S'$ or $'E'$.
<i>moder</i>	INTEGER. This specifies how the diagonal array <i>dr</i> is to be computed, just as <i>mode</i> specifies how <i>d</i> is to be computed.
<i>condr</i>	REAL for <i>slatmr</i> and <i>clatmr</i> , DOUBLE PRECISION for <i>dlatmr</i> and <i>zlatmr</i> , When <i>moder</i> is not zero, this specifies the condition number of the computed <i>dr</i> .
<i>pivtnng</i>	CHARACTER. On entry specifies pivoting permutations as follows: If <i>pivtnng</i> = $'N'$ or $' '$: no pivoting permutation. If <i>pivtnng</i> = $'L'$: left or row pivoting (matrix must be nonsymmetric). If <i>pivtnng</i> = $'R'$: right or column pivoting (matrix must be nonsymmetric). If <i>pivtnng</i> = $'B'$ or $'F'$: both or full pivoting, i.e., on both sides. In this case, <i>m</i> must equal <i>n</i> . If two calls to <i>?latmr</i> both have full bandwidth ($kl = m - 1$ and $ku = n - 1$), and differ only in the <i>pivtnng</i> and <i>pack</i> parameters, then the matrices generated differs only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be maintained with less than full bandwidth.
<i>ipivot</i>	INTEGER. Array, size (<i>n</i> or <i>m</i>) This array specifies the permutation used. After the basic matrix is generated, the rows, columns, or both are permuted.

If row pivoting is selected, `?latmr` starts with the last row and interchanges row m and row $ipivot(m)$, then moves to the next-to-last row, interchanging rows $(m-1)$ and row $ipivot(m-1)$, and so on. In terms of "2-cycles", the permutation is $(1\ ipivot(1))\ (2\ ipivot(2))\ \dots\ (m\ ipivot(m))$ where the rightmost cycle is applied first. This is the inverse of the effect of pivoting in LINPACK. The idea is that factoring (with pivoting) an identity matrix which has been inverse-pivoted in this way should result in a pivot vector identical to $ipivot$. Not referenced if $pivtnp = 'N'$.

sparse

REAL for `slatmr`,
 DOUBLE PRECISION for `dlatmr`,
 REAL for `clatmr`,
 DOUBLE PRECISION for `zlatmr`,

On entry, specifies the sparsity of the matrix if a sparse matrix is to be generated. *sparse* should lie between 0 and 1. To generate a sparse matrix, for each matrix entry a uniform (0, 1) random number x is generated and compared to *sparse*; if x is larger the matrix entry is unchanged and if x is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries is set to zero.

kl

INTEGER. On entry, specifies the lower bandwidth of the matrix. For example, $kl = 0$ implies upper triangular, $kl = 1$ implies upper Hessenberg, and kl at least $m-1$ implies the matrix is not banded. Must equal *ku* if matrix is symmetric or Hermitian.

ku

INTEGER. On entry, specifies the upper bandwidth of the matrix. For example, $ku = 0$ implies lower triangular, $ku = 1$ implies lower Hessenberg, and ku at least $n-1$ implies the matrix is not banded. Must equal *kl* if matrix is symmetric or Hermitian.

anorm

REAL for `slatmr`,
 DOUBLE PRECISION for `dlatmr`,
 REAL for `clatmr`,
 DOUBLE PRECISION for `zlatmr`,

On entry, specifies maximum entry of output matrix (output matrix is multiplied by a constant so that its largest absolute entry equal *anorm*) if *anorm* is nonnegative. If *anorm* is negative no scaling is done.

pack

for `slatmr`,
 for `dlatmr`,
 for `clatmr`,
 for `zlatmr`,

On entry, specifies packing of matrix as follows:

If *pack* = 'N': no packing

If *pack* = 'U': zero out all subdiagonal entries (if symmetric or Hermitian)

If *pack* = 'L': zero out all superdiagonal entries (if symmetric or Hermitian)

If *pack* = 'C': store the upper triangle columnwise (only if matrix symmetric or Hermitian or square upper triangular)

If *pack* = 'R': store the lower triangle columnwise (only if matrix symmetric or Hermitian or square lower triangular) (same as upper half rowwise if symmetric) (same as conjugate upper half rowwise if Hermitian)

If *pack* = 'B': store the lower triangle in band storage scheme (only if matrix symmetric or Hermitian)

If *pack* = 'Q': store the upper triangle in band storage scheme (only if matrix symmetric or Hermitian)

If *pack* = 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store *A* in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

LAPACK storage scheme	Value of <i>pack</i>
GB	'Z'
PB, HB or TB	'B' or 'Q'
PP, HP or TP	'C' or 'R'

If two calls to ?*latmr* differ only in the *pack* parameter, they generate mathematically equivalent matrices.

lda

INTEGER. On entry, *lda* specifies the first dimension of *a* as declared in the calling program.

If *pack* = 'N', 'U' or 'L', *lda* must be at least $\max(1, m)$.

If *pack* = 'C' or 'R', *lda* must be at least 1.

If *pack* = 'B', or 'Q', *lda* must be $\min(ku + 1, n)$.

If *pack* = 'Z', *lda* must be at least $k_{uu} + k_{ll} + 1$, where $k_{uu} = \min(ku, n-1)$ and $k_{ll} = \min(kl, n-1)$.

iwork

INTEGER. Array, size (*n* or *m*). Workspace. Not referenced if *pivtnng* = 'N'. Changed on exit.

Output Parameters

iseed

On exit, the seed is changed.

d

May be changed on exit if *mode* is nonzero.

dl

On exit, array is changed.

dr

On exit, array is changed.

a

REAL for *slatmr*,
DOUBLE PRECISION for *dlatmr*,

COMPLEX for `clatmr`,

DOUBLE COMPLEX for `zlatmr`,

On exit, *a* is the desired test matrix. Only those entries of *a* which are significant on output is referenced (even if *a* is in packed or band storage format). The unoccupied corners of *a* in band format are zeroed out.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -1, *m* is negative or unequal to *n* and *sym* = 'S' or 'H'.

If *info* = -2, *n* is negative .

If *info* = -3, *dist* is an illegal string.

If *info* = -5, *sym* is an illegal string..

If *info* = -7, *mode* is not in range -6 to 6.

If *info* = -8, *cond* is less than 1.0, and *mode* is neither -6, 0 nor 6.

If *info* = -10, *mode* is neither -6, 0 nor 6 and *rsign* is an illegal string.

If *info* = -11, *grade* is an illegal string, or *grade* = 'E' and *m* is not equal to *n*, or *grade*='L', 'R', 'B', 'S' or 'E' and *sym* = 'H', or *grade* = 'L', 'R', 'B', 'H' or 'E' and *sym* = 'S'

If *info* = -12, *grade* = 'E' and *dl* contains zero .

If *info* = -13, *model* is not in range -6 to 6 and *grade* = 'L', 'B', 'H', 'S' or 'E' .

If *info* = -14, *concl* is less than 1.0, *grade* = 'L', 'B', 'H', 'S' or 'E', and *model* is neither -6, 0 nor 6.

If *info* = -16, *moder* is not in range -6 to 6 and *grade* = 'R' or 'B' .

If *info* = -17, *condr* is less than 1.0, *grade* = 'R' or 'B', and *moder* is neither -6, 0 nor 6 .

If *info* = -18, *pivtnng* is an illegal string, or *pivtnng* = 'B' or 'F' and *m* is not equal to *n*, or *pivtnng* = 'L' or 'R' and *sym* = 'S' or 'H'.

If *info* = -19, *ipivot* contains out of range number and *pivtnng* is not equal to 'N' .

If *info* = -20, *kl* is negative.

If *info* = -21, *ku* is negative, or *sym* = 'S' or 'H' and *ku* not equal to *kl* .

If *info* = -22, *sparse* is not in range 0 to 1.

If *info* = -24, *pack* is an illegal string, or *pack* = 'U', 'L', 'B' or 'Q' and *sym* = 'N', or *pack* = 'C' and *sym* = 'N' and either *kl* is not equal to 0 or *n* is not equal to *m*, or *pack* = 'R' and *sym* = 'N', and either *ku* is not equal to 0 or *n* is not equal to *m* .

If *info* = -26, *lda* is too small .

If *info* = 1, error return from ?latm1 (computing *D*) .

If *info* = 2, cannot scale to *dmax* (max. entry is 0) .

If *info* = 3, error return from ?latm1(computing *dl*) .

If *info* = 4, error return from ?latm1(computing *dr*) .

If *info* = 5, *anorm* is positive, but matrix constructed prior to attempting to scale it to have norm *anorm*, is zero .

?latdf

Uses the LU factorization of the *n*-by-*n* matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.

Syntax

```
call slatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

```
call dlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

```
call clatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

```
call zlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpiv )
```

Include Files

- mkl.fi

Description

The routine ?latdf uses the LU factorization of the *n*-by-*n* matrix *Z* computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate by solving $Z^*x = b$ for *x*, and choosing the right-hand side *b* such that the norm of *x* is as large as possible. On entry *rhs* = *b* holds the contribution from earlier solved sub-systems, and on return *rhs* = *x*.

The factorization of *Z* returned by ?getc2 has the form $Z = P^*L^*U^*Q$, where *P* and *Q* are permutation matrices. *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

Input Parameters

<i>ijob</i>	INTEGER. <i>ijob</i> = 2: First compute an approximative null-vector <i>e</i> of <i>Z</i> using ?gecon, <i>e</i> is normalized, and solve for $Z^*x = \pm e - f$ with the sign giving the greater value of 2-norm(<i>x</i>). This option is about 5 times as expensive as default. <i>ijob</i> ≠ 2 (default): Local look ahead strategy where all entries of the right-hand side <i>b</i> is chosen as either +1 or -1 .
<i>n</i>	INTEGER. The number of columns of the matrix <i>Z</i> .
<i>z</i>	REAL for slatdf/clatdf DOUBLE PRECISION for dlatdf/zlatdf. Array, DIMENSION (<i>ldz</i> , <i>n</i>) On entry, the LU part of the factorization of the <i>n</i> -by- <i>n</i> matrix <i>Z</i> computed by ?getc2: $Z = P^*L^*U^*Q$.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>Z</i> . $lda \geq \max(1, n)$.

<i>rhs</i>	<p>REAL for slatdf/clatdf</p> <p>DOUBLE PRECISION for dlatdf/zlatdf.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>On entry, <i>rhs</i> contains contributions from other subsystems.</p>
<i>rdsum</i>	<p>REAL for slatdf/clatdf</p> <p>DOUBLE PRECISION for dlatdf/zlatdf.</p> <p>On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tgsyL, where the scaling factor <i>rdscal</i> has been factored out. If <i>trans</i> = 'T', <i>rdsum</i> is not touched.</p> <p>Note that <i>rdsum</i> only makes sense when ?tgsy2 is called by ?tgsyL.</p>
<i>rdscal</i>	<p>REAL for slatdf/clatdf</p> <p>DOUBLE PRECISION for dlatdf/zlatdf.</p> <p>On entry, scaling factor used to prevent overflow in <i>rdsum</i>.</p> <p>If <i>trans</i> = 'T', <i>rdscal</i> is not touched.</p> <p>Note that <i>rdscal</i> only makes sense when ?tgsy2 is called by ?tgsyL.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>The pivot indices; for $1 \leq i \leq n$, row <i>i</i> of the matrix has been interchanged with row <i>ipiv</i>(<i>i</i>).</p>
<i>jpiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>The pivot indices; for $1 \leq j \leq n$, column <i>j</i> of the matrix has been interchanged with column <i>jpiv</i>(<i>j</i>).</p>

Output Parameters

<i>rhs</i>	On exit, <i>rhs</i> contains the solution of the subsystem with entries according to the value of <i>ijob</i> .
<i>rdsum</i>	<p>On exit, the corresponding sum of squares updated with the contributions from the current sub-system.</p> <p>If <i>trans</i> = 'T', <i>rdsum</i> is not touched.</p>
<i>rdscal</i>	<p>On exit, <i>rdscal</i> is updated with respect to the current contributions in <i>rdsum</i>.</p> <p>If <i>trans</i> = 'T', <i>rdscal</i> is not touched.</p>

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call dlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call clatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call zlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
```

Include Files

- mkl.fi

Description

The routine ?latps solves one of the triangular systems

$$A*x = s*b, \text{ or } A^T*x = s*b, \text{ or } A^H*x = s*b \text{ (for complex flavors)}$$

with scaling to prevent overflow, where A is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem does not cause overflow, the Level 2 BLAS routine ?tpsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': upper triangular = 'L': uower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A*x = s*b$ (no transpose) = 'T': solve $A^T*x = s*b$ (transpose) = 'C': solve $A^H*x = s*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular. = 'N': non-unit triangular = 'U': unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> is set. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.

<i>ap</i>	<p>REAL for slatps DOUBLE PRECISION for dlatps COMPLEX for clatps DOUBLE COMPLEX for zlatps. Array, DIMENSION $(n(n+1)/2)$.</p> <p>The upper or lower triangular matrix A, packed columnwise in a linear array. The j-th column of A is stored in the array ap as follows: if $uplo = 'U'$, $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$; if $uplo = 'L'$, $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.</p>
<i>x</i>	<p>REAL for slatps DOUBLE PRECISION for dlatps COMPLEX for clatps DOUBLE COMPLEX for zlatps. Array, DIMENSION (n)</p> <p>On entry, the right hand side b of the triangular system.</p>
<i>cnorm</i>	<p>REAL for slatps/clatps DOUBLE PRECISION for dlatps/zlatps. Array, DIMENSION (n).</p> <p>If $normin = 'Y'$, $cnorm$ is an input argument and $cnorm(j)$ contains the norm of the off-diagonal part of the j-th column of A. If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or $'C'$, $cnorm(j)$ must be greater than or equal to the 1-norm.</p>
Output Parameters	
<i>x</i>	<p>On exit, x is overwritten by the solution vector x.</p>
<i>scale</i>	<p>REAL for slatps/clatps DOUBLE PRECISION for dlatps/zlatps.</p> <p>The scaling factor s for the triangular system as described above. If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $A*x = 0$.</p>
<i>cnorm</i>	<p>If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j-th column of A.</p>
<i>info</i>	<p>INTEGER. = 0: successful exit < 0: if $info = -k$, the k-th argument had an illegal value</p>

?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call slatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Include Files

- mkl.fi

Description

The routine ?latrd reduces nb rows and columns of a real symmetric or complex Hermitian matrix A to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation $Q^T * A * Q$ for real flavors, $Q^H * A * Q$ for complex flavors, and returns the matrices V and W which are needed to apply the transformation to the unreduced part of A .

If $uplo = 'U'$, ?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if $uplo = 'L'$, ?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by ?sytrd/?hetrd.

Input Parameters

$uplo$	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored: = 'U': upper triangular = 'L': lower triangular
n	INTEGER. The order of the matrix A .
nb	INTEGER. The number of rows and columns to be reduced.
a	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd DOUBLE COMPLEX for zlatrd. Array, DIMENSION (lda, n). On entry, the symmetric/Hermitian matrix A

If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER. The leading dimension of the array a . $lda \geq (1, n)$.

ldw INTEGER.

The leading dimension of the output array w . $ldw \geq \max(1, n)$.

Output Parameters

a On exit, if $uplo = 'U'$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of a ; the elements above the diagonal with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors;

if $uplo = 'L'$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of a ; the elements below the diagonal with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

e REAL for slatrd/clatrd
DOUBLE PRECISION for dlatrd/zlatrd.

If $uplo = 'U'$, $e(n-nb:n-1)$ contains the superdiagonal elements of the last nb columns of the reduced matrix;

if $uplo = 'L'$, $e(1:nb)$ contains the subdiagonal elements of the first nb columns of the reduced matrix.

tau REAL for slatrd
DOUBLE PRECISION for dlatrd
COMPLEX for clatrd
DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (lda, n) .

The scalar factors of the elementary reflectors, stored in $tau(n-nb:n-1)$ if $uplo = 'U'$, and in $tau(1:nb)$ if $uplo = 'L'$.

w REAL for slatrd
DOUBLE PRECISION for dlatrd
COMPLEX for clatrd
DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (ldw, n) .

The n -by- nb matrix W required to update the unreduced part of A .

Application Notes

If `uplo = 'U'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $a(1:i-1, i)$, and τ in $\tau(i-1)$.

If `uplo = 'L'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form $H(i) = I - \tau v v^T$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+1:n)$ is stored on exit in $a(i+1:n, i)$, and τ in $\tau(i)$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

$$A := A - VW^T - WV^T.$$

The contents of a on exit are illustrated by the following examples with $n = 5$ and $nb = 2$:

if <code>uplo = 'U'</code> :	if <code>uplo = 'L'</code> :
$\begin{bmatrix} a & a & a & v_1 & v_1 \\ & a & a & v_1 & v_1 \\ & & a & 1 & v_1 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$	$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_1 & a & a & \\ v_1 & v_1 & a & a & a \end{bmatrix}$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

slatrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call clatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Include Files

- `mkl.fi`

Description

The routine solves one of the triangular systems

$$A^*x = s*b, \text{ or } A^T*x = s*b, \text{ or } A^H*x = s*b \text{ (for complex flavors)}$$

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine `?trsv` is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A^*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A^*x = s*b$ (no transpose) = 'T': solve $A^T*x = s*b$ (transpose) = 'C': solve $A^H*x = s*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': non-unit triangular = 'U': unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. 0 On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$
<i>a</i>	REAL for <code>slatrs</code> DOUBLE PRECISION for <code>dlatrs</code> COMPLEX for <code>clatrs</code> DOUBLE COMPLEX for <code>zlatrs</code> . Array, DIMENSION (<i>lda</i> , <i>n</i>). Contains the triangular matrix A .

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *A* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *A* is not referenced.

If *diag* = 'U', the diagonal elements of *A* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

x REAL for slatrs
 DOUBLE PRECISION for dlatrs
 COMPLEX for clatrs
 DOUBLE COMPLEX for zlatrs.

Array, DIMENSION (*n*).

On entry, the right hand side *b* of the triangular system.

cnorm REAL for slatrs/clatrs
 DOUBLE PRECISION for dlatrs/zlatrs.

Array, DIMENSION (*n*).

If *normin* = 'Y', *cnorm* is an input argument and *cnorm* (*j*) contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm* (*j*) must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm*(*j*) must be greater than or equal to the 1-norm.

Output Parameters

x On exit, *x* is overwritten by the solution vector *x*.

scale REAL for slatrs/clatrs
 DOUBLE PRECISION for dlatrs/zlatrs.

Array, DIMENSION (*lda*, *n*). The scaling factor *s* for the triangular system as described above.

If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to $A*x = 0$.

cnorm If *normin* = 'N', *cnorm* is an output argument and *cnorm*(*j*) returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument had an illegal value

Application Notes

A rough bound on x is computed; if that is less than overflow, `?trsv` is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving $Ax = b$. The basic algorithm if A is lower triangular is

```
x[1:n] := b[1:n]
for j = 1, ..., n
x(j) := x(j) / A(j,j)
x[j+1:n] := x[j+1:n] - x(j)*a[j+1:n,j]
end
```

Define bounds on the components of x after j iterations of the loop:

```
M(j) = bound on x[1:j]
G(j) = bound on x[j+1:n]
```

Initially, let $M(0) = 0$ and $G(0) = \max\{x(i), i=1, \dots, n\}$.

Then for iteration $j+1$ we have

```
M(j+1) ≤ G(j) / | a(j+1,j+1) |
G(j+1) ≤ G(j) + M(j+1)*| a[j+2:n,j+1] |
≤ G(j) (1 + cnorm(j+1) / | a(j+1,j+1) |,
```

where $cnorm(j+1)$ is greater than or equal to the infinity-norm of column $j+1$ of a , not counting the diagonal. Hence

$$G(j) \leq G(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \leq (G(0)/|A(j,j)|) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \leq M(j)$, we use the Level 2 BLAS routine `?trsv` if the reciprocal of the largest $M(j)$, $j=1, \dots, n$, is larger than $\max(\text{underflow}, 1/\text{overflow})$.

The bound on $x(j)$ is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, x is scaled to prevent overflow, but if the bound overflows, x is set to 0, $x(j)$ to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^T x = b$ or $A^H x = b$. The basic algorithm for A upper triangular is

```
for j = 1, ..., n
x(j) := ( b(j) - A[1:j-1,j]' x[1:j-1] ) / A(j,j)
end
```

We simultaneously compute two bounds

$G(j)$ = bound on $(b(i) - A[1:i-1, i]' * x[1:i-1]), 1 \leq i \leq j$

$M(j)$ = bound on $x(i), 1 \leq i \leq j$

The initial values are $G(0) = 0, M(0) = \max\{b(i), i=1, \dots, n\}$, and we add the constraint $G(j) \geq G(j-1)$ and $M(j) \geq M(j-1)$ for $j \geq 1$.

Then the bound on $x(j)$ is

$M(j) \leq M(j-1) * (1 + cnorm(j)) / |A(j, j)|$

$$\leq M(0) \prod_{1 \leq i \leq j} (1 + cnorm(i) / |A(i, i)|)$$

and we can safely call ?trsv if $1/M(n)$ and $1/G(n)$ are both greater than $\max(\text{underflow}, 1/\text{overflow})$.

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

call slatz(*m, n, l, a, lda, tau, work*)

call dlatrz(*m, n, l, a, lda, tau, work*)

call clatz(*m, n, l, a, lda, tau, work*)

call zlatrz(*m, n, l, a, lda, tau, work*)

Include Files

- mkl.fi

Description

The routine ?latrz factors the m -by- $(m+l)$ real/complex upper trapezoidal matrix

$[A1 \ A2] = [A(1:m, 1:m) \ A(1:m, n-l+1:n)]$

as $(R \ 0) * Z$, by means of orthogonal/unitary transformations. Z is an $(m+l)$ -by- $(m+l)$ orthogonal/unitary matrix and R and $A1$ are m -by- m upper triangular matrices.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>l</i>	INTEGER. The number of columns of the matrix <i>A</i> containing the meaningful part of the Householder vectors. $n-m \geq l \geq 0$.
<i>a</i>	REAL for slatz DOUBLE PRECISION for dlatrz COMPLEX for clatz

DOUBLE COMPLEX for `zlatrz`.

Array, DIMENSION (*lda*, *n*).

On entry, the leading *m*-by-*n* upper trapezoidal part of the array *a* must contain the matrix to be factorized.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, m)$.

work REAL for `slatzr`

DOUBLE PRECISION for `dlatzr`

COMPLEX for `clatzr`

DOUBLE COMPLEX for `zlatrz`.

Workspace array, DIMENSION (*m*).

Output Parameters

a On exit, the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *n*-*l*+1 to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau REAL for `slatzr`

DOUBLE PRECISION for `dlatzr`

COMPLEX for `clatzr`

DOUBLE COMPLEX for `zlatrz`.

Array, DIMENSION (*m*).

The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - \tau u u^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - \tau a u^* u(k)^* T(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an l -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of A_2 .

The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of A_2 , such that the elements of $z(k)$ are in $a(k, l+1), \dots, a(k, n)$.

The elements of r are returned in the upper triangular part of A_1 .

Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

?lauu2

Computes the product $U*U^T(U*U^H)$ or $L^T*L(L^H*L)$, where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2( uplo, n, a, lda, info )
call dlauu2( uplo, n, a, lda, info )
call clauu2( uplo, n, a, lda, info )
call zlauu2( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?lauu2 computes the product $U*U^T$ or L^T*L for real flavors, and $U*U^H$ or L^H*L for complex flavors. Here the triangular factor U or L is stored in the upper or lower triangular part of the array a .

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in A .

If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in A .

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

$uplo$	CHARACTER*1. Specifies whether the triangular factor stored in the array a is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
n	INTEGER. The order of the triangular factor U or L . $n \geq 0$.

a REAL for slauu2
 DOUBLE PRECISION for dlauu2
 COMPLEX for clauu2
 DOUBLE COMPLEX for zlauu2.
 Array, DIMENSION (*lda*, *n*). On entry, the triangular factor *U* or *L*.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit,
 if *uplo* = 'U', then the upper triangle of *a* is overwritten with the upper triangle of the product U^*U^T (U^*U^H);
 if *uplo* = 'L', then the lower triangle of *a* is overwritten with the lower triangle of the product L^T*L (L^H*L).

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-k$, the *k*-th argument had an illegal value

?lauum

Computes the product $U^*U^T(U^*U^H)$ or L^T*L (L^H*L), where *U* and *L* are upper or lower triangular matrices (blocked algorithm).

Syntax

```
call slauum( uplo, n, a, lda, info )
call dlauum( uplo, n, a, lda, info )
call clauum( uplo, n, a, lda, info )
call zlauum( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?lauum computes the product U^*U^T or L^T*L for real flavors, and U^*U^H or L^H*L for complex flavors. Here the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the blocked form of the algorithm, calling [BLAS Level 3 Routines](#).

Input Parameters

The data types are given for the Fortran interface.

uplo CHARACTER*1.

Specifies whether the triangular factor stored in the array *a* is upper or lower triangular:

= 'U': Upper triangular

= 'L': Lower triangular

n INTEGER. The order of the triangular factor *U* or *L*. $n \geq 0$.

a REAL for `s1auum`
 DOUBLE PRECISION for `d1auum`
 COMPLEX for `c1auum`
 DOUBLE COMPLEX for `z1auum`.

Array of size (*lda*, *n*).

On entry, the triangular factor *U* or *L*.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit,
 if *uplo* = 'U', then the upper triangle of *a* is overwritten with the upper triangle of the product $U^*U^T(U^*U^H)$;
 if *uplo* = 'L', then the lower triangle of *a* is overwritten with the lower triangle of the product $L^T*L(L^H*L)$.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*k*, the *k*-th parameter had an illegal value.
 If *info* = -1011, memory allocation error occurred.

?orbdb1/?unbdb1

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call dorbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call cunbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call zunbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routines `?orbdb1/?unbdb1` simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} p_1 & | \\ \hline & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{21} is $(m - p)$ by q . q must not be larger than p , $m - p$, or $m - q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$	<code>?orbdb1/?unbdb1</code>
$p \leq \min(q, m - p, m - q)$	<code>?orbdb2/?unbdb2</code>
$m - p \leq \min(p, q, m - q)$	<code>?orbdb3/?unbdb3</code>
$m - q \leq \min(p, q, m - p)$	<code>?orbdb4/?unbdb4</code>

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m - p)$ -by- $(m - p)$, $(m - q)$ -by- $(m - q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of `?orcsd2by1/?uncsd2by1` for details on generating p_1 , p_2 , and q_1 using `?orgqr` and `?orglq`.

The upper-bidiagonal matrices b_{11} and b_{12} of size q by q are represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q - 1)$. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] or the description of `?orcsd/?uncsd` for details.

Input Parameters

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq \min(p, m - p, m - q)$.
x_{11}	REAL for <code>sorbdb1</code> DOUBLE PRECISION for <code>dorbdb1</code> COMPLEX for <code>cunbdb1</code> DOUBLE COMPLEX for <code>zunbdb1</code> Array, DIMENSION ($ldx11, q$). On entry, the top block of the orthogonal/unitary matrix to be reduced.
$ldx11$	INTEGER. The leading dimension of the array X_{11} . $ldx11 \geq p$.
x_{21}	REAL for <code>sorbdb1</code> DOUBLE PRECISION for <code>dorbdb1</code> COMPLEX for <code>cunbdb1</code>

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION ($ldx21, q$).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21

INTEGER. The leading dimension of the array X_{21} . $ldx21 \geq m-p$.

work

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Workspace array, DIMENSION ($lwork$).

lwork

INTEGER. The size of the *work* array. $lwork \geq m-q$

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11

The columns of $\text{tril}(x11)$ specify reflectors for p_1 and the rows of $\text{triu}(x11, 1)$ specify reflectors for q_1 , where $\text{tril}(A)$ denotes the lower triangle of A , and $\text{triu}(A)$ denotes the upper triangle of A .

x21

On exit, the columns of $\text{tril}(x21)$ specify the reflectors for p_2

theta

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2 REAL for sorbdb1
DOUBLE PRECISION for dorbdb1
COMPLEX for cunbdb1
DOUBLE COMPLEX for zunbdb1

Array, DIMENSION ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

tauq1 REAL for sorbdb1
DOUBLE PRECISION for dorbdb1
COMPLEX for cunbdb1
DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

info INTEGER.
= 0: successful exit
< 0: if *info* = $-i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

?orbdb2/?unbdb2

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
call dorbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
call cunbdb2( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

call `zunbdb2(m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work, lwork, info)`

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routines `?orbdb2/?unbdb2` simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} p_1 & | \\ \hline & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{12} is $(m - p)$ by q . q must not be larger than p , $m-p$, or $m-q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$	<code>?orbdb1/?unbdb1</code>
$p \leq \min(q, m - p, m - q)$	<code>?orbdb2/?unbdb2</code>
$m - p \leq \min(p, q, m - q)$	<code>?orbdb3/?unbdb3</code>
$m - q \leq \min(p, q, m - p)$	<code>?orbdb4/?unbdb4</code>

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of `?orcsd2by1/?uncsd2by1` for details on generating p_1 , p_2 , and q_1 using `?orgqr` and `?orglq`.

The upper-bidiagonal matrices b_{11} and b_{12} of size p by p are represented implicitly by angles $theta(1), \dots, theta(q)$ and $phi(1), \dots, phi(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of $theta$ with a sine or cosine of phi . See [Sutton09] or the description of `?orcsd/?uncsd` for details.

Input Parameters

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq \min(q, m-p, m-q)$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
$x11$	REAL for <code>sorbdb2</code> DOUBLE PRECISION for <code>dorbdb2</code> COMPLEX for <code>cunbdb2</code> DOUBLE COMPLEX for <code>zunbdb2</code> Array, DIMENSION ($ldx11, q$). On entry, the top block of the orthogonal/unitary matrix to be reduced.
$ldx11$	INTEGER. The leading dimension of the array X_{11} . $ldx11 \geq p$.
$x21$	REAL for <code>sorbdb2</code>

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION ($ldx21, q$).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21

INTEGER. The leading dimension of the array X_{21} . $ldx21 \geq m-p$.

work

REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Workspace array, DIMENSION ($lwork$).

lwork

INTEGER. The size of the *work* array. $lwork \geq m-q$

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11

On exit: the columns of $\text{tril}(x11)$ specify reflectors for p_1 and the rows of $\text{triu}(x11, 1)$ specify reflectors for q_1 .

x21

On exit, the columns of $\text{tril}(x21)$ specify the reflectors for p_2

theta

REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for cunbdb2

DOUBLE COMPLEX for zunbdb2

Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb2

DOUBLE PRECISION for dorbdb2

COMPLEX for `cunbdb2`

DOUBLE COMPLEX for `zunbdb2`

Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

`taup2`

REAL for `sorbdb2`

DOUBLE PRECISION for `dorbdb2`

COMPLEX for `cunbdb2`

DOUBLE COMPLEX for `zunbdb2`

Array, DIMENSION ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

`tauq1`

REAL for `sorbdb2`

DOUBLE PRECISION for `dorbdb2`

COMPLEX for `cunbdb2`

DOUBLE COMPLEX for `zunbdb2`

Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

`info`

INTEGER.

= 0: successful exit

< 0: if `info` = $-i$, the i -th argument has an illegal value.

See Also

[?orcscd/?uncscd](#)

[?orcscd2by1/?uncscd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

[?orbdb3/?unbdb3](#)

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

```
call sorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call dorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call cunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

```
call zunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines `?orbdb3/?unbdb3` simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} p_1 & | & \\ \hline & & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{12} is $(m - p)$ by q . $m - p$ must not be larger than p , q , or $m - q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$	<code>?orbdb1/?unbdb1</code>
$p \leq \min(q, m - p, m - q)$	<code>?orbdb2/?unbdb2</code>
$m - p \leq \min(p, q, m - q)$	<code>?orbdb3/?unbdb3</code>
$m - q \leq \min(p, q, m - p)$	<code>?orbdb4/?unbdb4</code>

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of `?orcsd2by1/?uncsd2by1` for details on generating p_1 , p_2 , and q_1 using `?orgqr` and `?orglq`.

The upper-bidiagonal matrices b_{11} and b_{12} of size $(m-p)$ by $(m-p)$ are represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] or the description of `?orcsd/?uncsd` for details.

Input Parameters

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$, $m - p \leq \min(p, q, m - q)$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
$x11$	REAL for <code>sorbdb3</code> DOUBLE PRECISION for <code>dorbdb3</code> COMPLEX for <code>cunbdb3</code> DOUBLE COMPLEX for <code>zunbdb3</code> Array, DIMENSION $(ldx11, q)$. On entry, the top block of the orthogonal/unitary matrix to be reduced.

<i>ldx11</i>	INTEGER. The leading dimension of the array X_{11} . $ldx11 \geq p$.
<i>x21</i>	REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Array, DIMENSION ($ldx21, q$). On entry, the bottom block of the orthogonal/unitary matrix to be reduced.
<i>ldx21</i>	INTEGER. The leading dimension of the array X_{21} . $ldx21 \geq m-p$.
<i>work</i>	REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Workspace array, DIMENSION ($lwork$).
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. $lwork \geq m-q$ If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.

Output Parameters

<i>x11</i>	On exit: the columns of <code>tril(x11)</code> specify reflectors for p_1 and the rows of <code>triu(x11,1)</code> specify reflectors for q_1 .
<i>x21</i>	On exit, the columns of <code>tril(x21)</code> specify the reflectors for p_2
<i>theta</i>	REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles <i>theta</i> and <i>phi</i> . See the Description section for details.
<i>phi</i>	REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles <i>theta</i> and <i>phi</i> . See the Description section for details.

taup1 REAL for sorbdb3
DOUBLE PRECISION for dorbdb3
COMPLEX for cunbdb3
DOUBLE COMPLEX for zunbdb3
Array, DIMENSION (p).
Scalar factors of the elementary reflectors that define p_1 .

taup2 REAL for sorbdb3
DOUBLE PRECISION for dorbdb3
COMPLEX for cunbdb3
DOUBLE COMPLEX for zunbdb3
Array, DIMENSION ($m-p$).
Scalar factors of the elementary reflectors that define p_2 .

tauq1 REAL for sorbdb3
DOUBLE PRECISION for dorbdb3
COMPLEX for cunbdb3
DOUBLE COMPLEX for zunbdb3
Array, DIMENSION (q).
Scalar factors of the elementary reflectors that define q_1 .

info INTEGER.
= 0: successful exit
< 0: if *info* = $-i$, the i -th argument has an illegal value.

See Also[?orcsd/?uncsd](#)[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.[xerbla](#)**[?orbdb4/?unbdb4](#)***Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.***Syntax**

```
call sorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
```

```
call dorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
```

```
call cunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
```

```
call zunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The routines ?orbdb4/?unbdb4 simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} p_1 & | & \\ \hline & & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{21} is $(m - p)$ by q . $m - q$ must not be larger than q , p , or $m - p$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m - p)$ -by- $(m - p)$, $(m - q)$ -by- $(m - q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1 , p_2 , and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size $(m - q)$ by $(m - q)$ are represented implicitly by angles $theta(1), \dots, theta(q)$ and $phi(1), \dots, phi(q - 1)$. Every entry in each bidiagonal band is a product of a sine or cosine of $theta$ with a sine or cosine of phi . See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$ and $0 \leq m - q \leq \min(p, m - p, q)$.
$x11$	REAL for sorbdb4 DOUBLE PRECISION for dorbdb4 COMPLEX for cunbdb4 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($ldx11, q$).

On entry, the top block of the orthogonal/unitary matrix to be reduced.

ldx11 INTEGER. The leading dimension of the array X_{11} . $ldx11 \geq p$.

x21 REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($ldx21, q$).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

ldx21 INTEGER. The leading dimension of the array X_{21} . $ldx21 \geq m-p$.

work REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Workspace array, DIMENSION ($lwork$).

lwork INTEGER. The size of the *work* array. $lwork \geq m-q$

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11 On exit: the columns of $\text{tril}(x11)$ specify reflectors for p_1 and the rows of $\text{triu}(x11, 1)$ specify reflectors for q_1 .

x21 On exit, the columns of $\text{tril}(x21)$ specify the reflectors for p_2

theta REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles $theta$ and phi . See the Description section for details.

taup1

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

tauq1

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

phantom

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (m).

The routine computes an m -by-1 column vector y that is orthogonal to the columns of $[x_{11}; x_{21}]$. $phantom(1:p)$ and $phantom(p+1:m)$ contain Householder vectors for $y(1:p)$ and $y(p+1:m)$, respectively.

info

INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#)

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.
xerbla

[?orbdb5/?unbdb5](#)

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

```
call sorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call dorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call cunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call zunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- mkl.fi, lapack.f90

Description

The [?orbdb5/?unbdb5](#) routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then some other vector from the orthogonal complement is returned. This vector is chosen in an arbitrary but deterministic way.

Input Parameters

<i>m1</i>	INTEGER	The dimension of <i>x1</i> and the number of rows in <i>q1</i> . $0 \leq m1$.
<i>m2</i>	INTEGER	The dimension of <i>x2</i> and the number of rows in <i>q2</i> . $0 \leq m2$.
<i>n</i>	INTEGER	The number of columns in <i>q1</i> and <i>q2</i> . $0 \leq n$.
<i>x1</i>	REAL for sorbdb5	

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size $m1$.

The top part of the vector to be orthogonalized.

incx1

INTEGER

Increment for entries of $x1$.

$x2$

REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size $m2$.

The bottom part of the vector to be orthogonalized.

incx2

INTEGER

Increment for entries of $x2$.

$q1$

REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size $(ldq1, n)$.

The top part of the orthonormal basis matrix.

$ldq1$

INTEGER

The leading dimension of $q1$. $ldq1 \geq m1$.

$q2$

REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for zundb5

Array of size $(ldq2, n)$.

The bottom part of the orthonormal basis matrix.

$ldq2$

INTEGER

The leading dimension of $q2$. $ldq2 \geq m2$.

work

REAL for sordb5

DOUBLE PRECISION for dordb5

COMPLEX for cundb5

COMPLEX*16 for `zunbdb5`

Workspace array of size `lwork`.

`lwork`

INTEGER

The size of the array `work`. $lwork \geq n$.

Output Parameters

`x1`

The top part of the projected vector.

`x2`

The bottom part of the projected vector.

`info`

INTEGER.

= 0: successful exit

< 0: if `info = -i`, the *i*-th argument has an illegal value.

See Also

[?orcsd/?uncsd](#)

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

[?orbdb6/?unbdb6](#)

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

```
call sorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call dorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call cunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

```
call zunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The [?orbdb6/?unbdb6](#) routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then the zero vector is returned.

Input Parameters

<i>m1</i>	INTEGER The dimension of <i>x1</i> and the number of rows in <i>q1</i> . $0 \leq m1$.
<i>m2</i>	INTEGER The dimension of <i>x2</i> and the number of rows in <i>q2</i> . $0 \leq m2$.
<i>n</i>	INTEGER The number of columns in <i>q1</i> and <i>q2</i> . $0 \leq n$.
<i>x1</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size <i>m1</i> . The top part of the vector to be orthogonalized.
<i>incx1</i>	INTEGER Increment for entries of <i>x1</i> .
<i>x2</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size <i>m2</i> . The bottom part of the vector to be orthogonalized.
<i>incx2</i>	INTEGER Increment for entries of <i>x2</i> .
<i>q1</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size (<i>ldq1</i> , <i>n</i>). The top part of the orthonormal basis matrix.

<i>ldq1</i>	INTEGER The leading dimension of <i>q1</i> . $ldq1 \geq m1$.
<i>q2</i>	REAL for <i>soradb5</i> DOUBLE PRECISION for <i>doradb5</i> COMPLEX for <i>cunadb5</i> COMPLEX*16 for <i>zunadb5</i> Array of size (<i>ldq2</i> , <i>n</i>). The bottom part of the orthonormal basis matrix.
<i>ldq2</i>	INTEGER The leading dimension of <i>q2</i> . $ldq2 \geq m2$.
<i>work</i>	REAL for <i>soradb5</i> DOUBLE PRECISION for <i>doradb5</i> COMPLEX for <i>cunadb5</i> COMPLEX*16 for <i>zunadb5</i> Workspace array of size <i>lwork</i> .
<i>lwork</i>	INTEGER The size of the array <i>work</i> . $lwork \geq n$.

Output Parameters

<i>x1</i>	The top part of the projected vector.
<i>x2</i>	The bottom part of the projected vector.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = <i>-i</i> , the <i>i</i> -th argument has an illegal value.

See Also

[?orcscd/?uncscd](#)

[?orcscd2by1/?uncscd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorg2l( m, n, k, a, lda, tau, work, info )
call dorg2l( m, n, k, a, lda, tau, work, info )
call cung2l( m, n, k, a, lda, tau, work, info )
call zung2l( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?org2l/?ung2l generates an m -by- n real/complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$Q = H(k) * \dots * H(2) * H(1)$ as returned by ?geqlf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q . $m \geq n \geq 0$.
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
a	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (lda, n). On entry, the $(n-k+i)$ -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqlf in the last k columns of its array argument A .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
τ	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (k). $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqlf.

work REAL for sorg2l
 DOUBLE PRECISION for dorg2l
 COMPLEX for cung2l
 DOUBLE COMPLEX for zung2l.
 Workspace array, DIMENSION (*n*).

Output Parameters

a On exit, the *m*-by-*n* matrix *Q*.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix *Q* from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorg2r( m, n, k, a, lda, tau, work, info )
call dorg2r( m, n, k, a, lda, tau, work, info )
call cung2r( m, n, k, a, lda, tau, work, info )
call zung2r( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?org2r/?ung2r generates an *m*-by-*n* real/complex matrix *Q* with orthonormal columns, which is defined as the first *n* columns of a product of *k* elementary reflectors of order *m*

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by ?geqrf.

Input Parameters

m INTEGER. The number of rows of the matrix *Q*. $m \geq 0$.

n INTEGER. The number of columns of the matrix *Q*. $m \geq n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. $n \geq k \geq 0$.

a REAL for sorg2r
 DOUBLE PRECISION for dorg2r

COMPLEX for `cung2r`
 DOUBLE COMPLEX for `zung2r`.

Array, DIMENSION (*lda*, *n*).

On entry, the *i*-th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `?geqrf` in the first *k* columns of its array argument *a*.

lda INTEGER. The first DIMENSION of the array *a*. $lda \geq \max(1, m)$.

tau REAL for `sorg2r`
 DOUBLE PRECISION for `dorg2r`
 COMPLEX for `cung2r`
 DOUBLE COMPLEX for `zung2r`.

Array, DIMENSION (*k*).

tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?geqrf`.

work REAL for `sorg2r`
 DOUBLE PRECISION for `dorg2r`
 COMPLEX for `cung2r`
 DOUBLE COMPLEX for `zung2r`.
 Workspace array, DIMENSION (*n*).

Output Parameters

a On exit, the *m*-by-*n* matrix *Q*.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix *Q* from an LQ factorization determined by `?gelqf` (unblocked algorithm).

Syntax

```
call sorgl2( m, n, k, a, lda, tau, work, info )
call dorgl2( m, n, k, a, lda, tau, work, info )
call cungl2( m, n, k, a, lda, tau, work, info )
call zungl2( m, n, k, a, lda, tau, work, info )
```

Include Files

- `mkl.fi`

Description

The routine `?orgl2/?ungl2` generates a m -by- n real/complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^{H*} \dots (H(2))^{H*} (H(1))^{H*}$ for complex flavors as returned by `?gelqf`.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix Q . $n \geq m$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
<i>a</i>	REAL for <code>sorgl2</code> DOUBLE PRECISION for <code>dorgl2</code> COMPLEX for <code>cungl2</code> DOUBLE COMPLEX for <code>zungl2</code> . Array, DIMENSION (lda, n). On entry, the i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?gelqf</code> in the first k rows of its array argument a .
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for <code>sorgl2</code> DOUBLE PRECISION for <code>dorgl2</code> COMPLEX for <code>cungl2</code> DOUBLE COMPLEX for <code>zungl2</code> . Array, DIMENSION (k). $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>?gelqf</code> .
<i>work</i>	REAL for <code>sorgl2</code> DOUBLE PRECISION for <code>dorgl2</code> COMPLEX for <code>cungl2</code> DOUBLE COMPLEX for <code>zungl2</code> . Workspace array, DIMENSION (m).

Output Parameters

<i>a</i>	On exit, the m -by- n matrix Q .
<i>info</i>	INTEGER. = 0: successful exit < 0: if $info = -i$, the i -th argument has an illegal value.

?org2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).

Syntax

```
call sorg2( m, n, k, a, lda, tau, work, info )
call dorgr2( m, n, k, a, lda, tau, work, info )
call cungr2( m, n, k, a, lda, tau, work, info )
call zungr2( m, n, k, a, lda, tau, work, info )
```

Include Files

- mkl.fi

Description

The routine ?org2/?ungr2 generates an m -by- n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or $Q = (H(1))^{H*} (H(2))^{H*} \dots (H(k))^{H*}$ for complex flavors as returned by ?gerqf.

Input Parameters

m	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n	INTEGER. The number of columns of the matrix Q . $n \geq m$
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
a	REAL for sorg2 DOUBLE PRECISION for dorgr2 COMPLEX for cungr2 DOUBLE COMPLEX for zungr2. Array, DIMENSION (lda, n). On entry, the ($m - k + i$)-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gerqf in the last k rows of its array argument a .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
tau	REAL for sorg2 DOUBLE PRECISION for dorgr2 COMPLEX for cungr2 DOUBLE COMPLEX for zungr2.

Array, DIMENSION (k). $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?gerqf`.

work REAL for `sorghr2`
 DOUBLE PRECISION for `dorghr2`
 COMPLEX for `cungr2`
 DOUBLE COMPLEX for `zungr2`.
 Workspace array, DIMENSION (m).

Output Parameters

a On exit, the m -by- n matrix Q .
info INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- `mk1.fi`

Description

The routine `?orm2l/?unm2l` overwrites the general real/complex m -by- n matrix C with

$Q^T C$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T C / Q^H C$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$C Q$ if $side = 'R'$ and $trans = 'N'$, or

$C Q^T / C Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by `?geqlf`.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side CHARACTER*1.

= 'L': apply Q or Q^T / Q^H from the left
 = 'R': apply Q or Q^T / Q^H from the right

trans

CHARACTER*1.
 = 'N': apply Q (no transpose)
 = 'T': apply Q^T (transpose, for real flavors)
 = 'C': apply Q^H (conjugate transpose, for complex flavors)

m

INTEGER. The number of rows of the matrix C . $m \geq 0$.

n

INTEGER. The number of columns of the matrix C . $n \geq 0$.

k

INTEGER. The number of elementary reflectors whose product defines the matrix Q .

If *side* = 'L', $m \geq k \geq 0$;

if *side* = 'R', $n \geq k \geq 0$.

a

REAL for *sorm21*
 DOUBLE PRECISION for *dorm21*
 COMPLEX for *cunm21*
 DOUBLE COMPLEX for *zunm21*.

Array, DIMENSION (*lda*,*k*).

The *i*-th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `?geqlf` in the last *k* columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda

INTEGER. The leading dimension of the array *a*.

If *side* = 'L', $lda \geq \max(1, m)$

if *side* = 'R', $lda \geq \max(1, n)$.

tau

REAL for *sorm21*
 DOUBLE PRECISION for *dorm21*
 COMPLEX for *cunm21*
 DOUBLE COMPLEX for *zunm21*.

Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?geqlf`.

c

REAL for *sorm21*
 DOUBLE PRECISION for *dorm21*
 COMPLEX for *cunm21*
 DOUBLE COMPLEX for *zunm21*.

Array, DIMENSION (*ldc*, *n*).

On entry, the *m*-by-*n* matrix C .

ldc INTEGER. The leading dimension of the array *C*. $ldc \geq \max(1, m)$.

work REAL for *sorm2l*
 DOUBLE PRECISION for *dorm2l*
 COMPLEX for *cunm2l*
 DOUBLE COMPLEX for *zunm2l*.

Workspace array, DIMENSION:
 (*n*) if *side* = 'L',
 (*m*) if *side* = 'R'.

Output Parameters

c On exit, *c* is overwritten by Q^*C or Q^T*C / Q^H*C , or $C*Q$, or $C*Q^T$ / $C*Q^H$.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?orm2r/?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?orm2r/?unm2r overwrites the general real/complex *m*-by-*n* matrix *C* with

Q^*C if *side* = 'L' and *trans* = 'N', or

Q^T*C / Q^H*C if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

$C*Q$ if *side* = 'R' and *trans* = 'N', or

$C*Q^T$ / $C*Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here *Q* is a real orthogonal or complex unitary matrix defined as the product of *k* elementary reflectors

$Q = H(1)*H(2)*\dots*H(k)$ as returned by ?geqrf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

<i>side</i>	<p>CHARACTER*1.</p> <p>= 'L': apply Q or Q^T / Q^H from the left</p> <p>= 'R': apply Q or Q^T / Q^H from the right</p>
<i>trans</i>	<p>CHARACTER*1.</p> <p>= 'N': apply Q (no transpose)</p> <p>= 'T': apply Q^T (transpose, for real flavors)</p> <p>= 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q.</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$;</p> <p>if <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>REAL for <code>sorm2r</code></p> <p>DOUBLE PRECISION for <code>dorm2r</code></p> <p>COMPLEX for <code>cunm2r</code></p> <p>DOUBLE COMPLEX for <code>zunm2r</code>.</p> <p>Array, DIMENSION (<i>lda</i>,<i>k</i>).</p> <p>The <i>i</i>-th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?geqrf</code> in the first <i>k</i> columns of its array argument <i>a</i>. The array <i>a</i> is modified by the routine but restored on exit.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>.</p> <p>If <i>side</i> = 'L', $lda \geq \max(1, m)$;</p> <p>if <i>side</i> = 'R', $lda \geq \max(1, n)$.</p>
<i>tau</i>	<p>REAL for <code>sorm2r</code></p> <p>DOUBLE PRECISION for <code>dorm2r</code></p> <p>COMPLEX for <code>cunm2r</code></p> <p>DOUBLE COMPLEX for <code>zunm2r</code>.</p> <p>Array, DIMENSION (<i>k</i>).</p> <p><i>tau</i>(<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>?geqrf</code>.</p>
<i>c</i>	<p>REAL for <code>sorm2r</code></p> <p>DOUBLE PRECISION for <code>dorm2r</code></p> <p>COMPLEX for <code>cunm2r</code></p>

DOUBLE COMPLEX for `zunm2r`.

Array, DIMENSION (ldc, n).

On entry, the m -by- n matrix C .

`ldc` INTEGER. The leading dimension of the array `c`. $ldc \geq \max(1, m)$.

`work` REAL for `sorm2r`
 DOUBLE PRECISION for `dorm2r`
 COMPLEX for `cunm2r`
 DOUBLE COMPLEX for `zunm2r`.

Workspace array, DIMENSION

(n) if `side = 'L'`,

(m) if `side = 'R'`.

Output Parameters

`c` On exit, `c` is overwritten by Q^*C or $Q^T C$ / $Q^H C$, or C^*Q , or C^*Q^T / C^*Q^H .

`info` INTEGER.
 = 0: successful exit
 < 0: if `info = -i`, the i -th argument had an illegal value

?orml2/?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).

Syntax

`call sorml2(side, trans, m, n, k, a, lda, tau, c, ldc, work, info)`

`call dorml2(side, trans, m, n, k, a, lda, tau, c, ldc, work, info)`

`call cunml2(side, trans, m, n, k, a, lda, tau, c, ldc, work, info)`

`call zunml2(side, trans, m, n, k, a, lda, tau, c, ldc, work, info)`

Include Files

- `mk1.fi`

Description

The routine `?orml2/?unml2` overwrites the general real/complex m -by- n matrix C with

Q^*C if `side = 'L'` and `trans = 'N'`, or

$Q^T C$ / $Q^H C$ if `side = 'L'` and `trans = 'T'` (for real flavors) or `trans = 'C'` (for complex flavors), or

C^*Q if `side = 'R'` and `trans = 'N'`, or

C^*Q^T / C^*Q^H if `side = 'R'` and `trans = 'T'` (for real flavors) or `trans = 'C'` (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^{H*} \dots (H(2))^{H*} (H(1))^{H*}$ for complex flavors as returned by ?gelqf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	<p>CHARACTER*1.</p> <p>= 'L': apply Q or Q^T / Q^H from the left</p> <p>= 'R': apply Q or Q^T / Q^H from the right</p>
<i>trans</i>	<p>CHARACTER*1.</p> <p>= 'N': apply Q (no transpose)</p> <p>= 'T': apply Q^T (transpose, for real flavors)</p> <p>= 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q.</p> <p>If $side = 'L'$, $m \geq k \geq 0$;</p> <p>if $side = 'R'$, $n \geq k \geq 0$.</p>
<i>a</i>	<p>REAL for sorml2</p> <p>DOUBLE PRECISION for dorml2</p> <p>COMPLEX for cunml2</p> <p>DOUBLE COMPLEX for zunml2.</p> <p>Array, DIMENSION</p> <p>(<i>lda</i>, <i>m</i>) if $side = 'L'$,</p> <p>(<i>lda</i>, <i>n</i>) if $side = 'R'$</p> <p>The i-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gelqf in the first k rows of its array argument a. The array a is modified by the routine but restored on exit.</p>
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, k)$.
<i>tau</i>	<p>REAL for sorml2</p> <p>DOUBLE PRECISION for dorml2</p> <p>COMPLEX for cunml2</p> <p>DOUBLE COMPLEX for zunml2.</p> <p>Array, DIMENSION (k).</p> <p>$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gelqf.</p>

c REAL for `sorml2`
DOUBLE PRECISION for `dorml2`
COMPLEX for `cunml2`
DOUBLE COMPLEX for `zunml2`.
Array, DIMENSION (*ldc*, *n*) On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. $ldc \geq \max(1, m)$.

work REAL for `sorml2`
DOUBLE PRECISION for `dorml2`
COMPLEX for `cunml2`
DOUBLE COMPLEX for `zunml2`.
Workspace array, DIMENSION
(*n*) if *side* = 'L',
(*m*) if *side* = 'R'

Output Parameters

c On exit, *c* is overwritten by Q^*C or $Q^T C$ / $Q^H C$, or C^*Q , or C^*Q^T / C^*Q^H .

info INTEGER.
= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr2/?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).

Syntax

```
call sormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- mkl.fi

Description

The routine ?ormr2/?unmr2 overwrites the general real/complex *m*-by-*n* matrix *C* with

Q^*C if *side* = 'L' and *trans* = 'N', or

$Q^T C$ / $Q^H C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

C^*Q if *side* = 'R' and *trans* = 'N', or

C^*Q^T / C^*Q^H if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or $Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by `?gerqf`.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	<p>CHARACTER*1.</p> <p>= 'L': apply Q or Q^T / Q^H from the left</p> <p>= 'R': apply Q or Q^T / Q^H from the right</p>
<i>trans</i>	<p>CHARACTER*1.</p> <p>= 'N': apply Q (no transpose)</p> <p>= 'T': apply Q^T (transpose, for real flavors)</p> <p>= 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	<p>INTEGER. The number of elementary reflectors whose product defines the matrix Q.</p> <p>If $side = 'L'$, $m \geq k \geq 0$;</p> <p>if $side = 'R'$, $n \geq k \geq 0$.</p>
<i>a</i>	<p>REAL for <code>sormr2</code></p> <p>DOUBLE PRECISION for <code>dormr2</code></p> <p>COMPLEX for <code>cunmr2</code></p> <p>DOUBLE COMPLEX for <code>zunmr2</code>.</p> <p>Array, DIMENSION</p> <p>(<i>lda</i>, <i>m</i>) if $side = 'L'$,</p> <p>(<i>lda</i>, <i>n</i>) if $side = 'R'$</p> <p>The i-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?gerqf</code> in the last k rows of its array argument a. The array a is modified by the routine but restored on exit.</p>
<i>lda</i>	<p>INTEGER.</p> <p>The leading dimension of the array a. $lda \geq \max(1, k)$.</p>
<i>tau</i>	<p>REAL for <code>sormr2</code></p> <p>DOUBLE PRECISION for <code>dormr2</code></p> <p>COMPLEX for <code>cunmr2</code></p> <p>DOUBLE COMPLEX for <code>zunmr2</code>.</p> <p>Array, DIMENSION (k).</p>

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?gerqf`.

`c` REAL for `sormr2`
 DOUBLE PRECISION for `dormr2`
 COMPLEX for `cunmr2`
 DOUBLE COMPLEX for `zunmr2`.
 Array, DIMENSION (l_{dc} , n).
 On entry, the m -by- n matrix C .

`l_{dc}` INTEGER. The leading dimension of the array `c`. $l_{dc} \geq \max(1, m)$.

`work` REAL for `sormr2`
 DOUBLE PRECISION for `dormr2`
 COMPLEX for `cunmr2`
 DOUBLE COMPLEX for `zunmr2`.
 Workspace array, DIMENSION
 (n) if `side = 'L'`,
 (m) if `side = 'R'`

Output Parameters

`c` On exit, `c` is overwritten by Q^*C or $Q^T C$ / $Q^{H*}C$, or C^*Q , or C^*Q^T / C^*Q^H .

`info` INTEGER.
 = 0: successful exit
 < 0: if `info = -i`, the i -th argument had an illegal value

`?ormr3/?unmr3`

Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by `?tzzrf` (unblocked algorithm).

Syntax

```
call sormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call dormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call cunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call zunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
```

Include Files

- `mkl.fi`

Description

The routine `?ormr3/?unmr3` overwrites the general real/complex m -by- n matrix C with

Q^*C if $side = 'L'$ and $trans = 'N'$, or

Q^T*C / Q^H*C if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$C*Q$ if $side = 'R'$ and $trans = 'N'$, or

$C*Q^T / C*Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1)*H(2)*\dots*H(k)$ as returned by `?tzzrf`.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. If $side = 'L'$, $m \geq l \geq 0$, if $side = 'R'$, $n \geq l \geq 0$.
<i>a</i>	REAL for <code>sormr3</code> DOUBLE PRECISION for <code>dormr3</code> COMPLEX for <code>cunmr3</code> DOUBLE COMPLEX for <code>zunmr3</code> . Array, DIMENSION (<i>lda</i> , <i>m</i>) if $side = 'L'$, (<i>lda</i> , <i>n</i>) if $side = 'R'$ The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?tzzrf</code> in the last k rows of its array argument a . The array a is modified by the routine but restored on exit.

<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, k)$.
<i>tau</i>	REAL for <i>sormr3</i> DOUBLE PRECISION for <i>dormr3</i> COMPLEX for <i>cunmr3</i> DOUBLE COMPLEX for <i>zunmr3</i> . Array, DIMENSION (<i>k</i>). <i>tau(i)</i> must contain the scalar factor of the elementary reflector $H(i)$, as returned by <i>?tzzrf</i> .
<i>c</i>	REAL for <i>sormr3</i> DOUBLE PRECISION for <i>dormr3</i> COMPLEX for <i>cunmr3</i> DOUBLE COMPLEX for <i>zunmr3</i> . Array, DIMENSION (<i>ldc</i> , <i>n</i>). On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$.
<i>work</i>	REAL for <i>sormr3</i> DOUBLE PRECISION for <i>dormr3</i> COMPLEX for <i>cunmr3</i> DOUBLE COMPLEX for <i>zunmr3</i> . Workspace array, DIMENSION (<i>n</i>) if <i>side</i> = 'L', (<i>m</i>) if <i>side</i> = 'R'.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by Q^*C or Q^T*C / $Q^{H*}C$, or $C*Q$, or $C*Q^T$ / $C*Q^H$.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value

?pbtf2

Computes the Cholesky factorization of a symmetric/Hermitian positive-definite band matrix (unblocked algorithm).

Syntax

```
call spbtf2( uplo, n, kd, ab, ldab, info )
call dpbtf2( uplo, n, kd, ab, ldab, info )
```

```
call cpbtf2( uplo, n, kd, ab, ldab, info )
```

```
call zpbtf2( uplo, n, kd, ab, ldab, info )
```

Include Files

- mkl.fi

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix A .

The factorization has the form

$A = U^T * U$ for real flavors, $A = U^H * U$ for complex flavors if $uplo = 'U'$, or

$A = L * L^T$ for real flavors, $A = L * L^H$ for complex flavors if $uplo = 'L'$,

where U is an upper triangular matrix, and L is lower triangular. This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix A is stored: = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix A if $uplo = 'U'$, or the number of sub-diagonals if $uplo = 'L'$. $kd \geq 0$.
<i>ab</i>	REAL for spbtf2 DOUBLE PRECISION for dpbtf2 COMPLEX for cpbtf2 DOUBLE COMPLEX for zpbtf2. Array, DIMENSION ($ldab, n$). On entry, the upper or lower triangle of the symmetric/ Hermitian band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array ab as follows: if $uplo = 'U'$, $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$; if $uplo = 'L'$, $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.
<i>ldab</i>	INTEGER. The leading dimension of the array ab . $ldab \geq kd+1$.

Output Parameters

<i>ab</i>	On exit, If <i>info</i> = 0, the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^T*U$ ($A=U^H*U$), or $A= L*L^T$ ($A = L*L^H$) of the band matrix <i>A</i> , in the same storage format as <i>A</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th argument had an illegal value > 0: if <i>info</i> = <i>k</i> , the leading minor of order <i>k</i> is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive-definite matrix (unblocked algorithm).

Syntax

```
call spotf2( uplo, n, a, lda, info )
call dpotf2( uplo, n, a, lda, info )
call cpotf2( uplo, n, a, lda, info )
call zpotf2( uplo, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?potf2 computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix *A*. The factorization has the form

$A = U^T*U$ for real flavors, $A = U^H*U$ for complex flavors if *uplo* = 'U', or

$A = L*L^T$ for real flavors, $A = L*L^H$ for complex flavors if *uplo* = 'L',

where *U* is an upper triangular matrix, and *L* is lower triangular.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#)

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix <i>A</i> is stored. = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	REAL for spotf2 DOUBLE PRECISION or dpotf2

COMPLEX for `cpotf2`
 DOUBLE COMPLEX for `zpotf2`.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*.
lda ≥ max(1, *n*).

Output Parameters

a On exit, If *info* = 0, the factor *U* or *L* from the Cholesky factorization $A=U^T*U$ ($A=U^H*U$), or $A= L*L^T$ ($A = L*L^H$).

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*k*, the *k*-th argument had an illegal value
 > 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?ptts2

Solves a tridiagonal system of the form $A*X=B$ using the $L*D*L^H/L*D*L^H$ factorization computed by `?pttrf`.

Syntax

```
call sptts2( n, nrhs, d, e, b, ldb )
call dptts2( n, nrhs, d, e, b, ldb )
call cptts2( iuplo, n, nrhs, d, e, b, ldb )
call zptts2( iuplo, n, nrhs, d, e, b, ldb )
```

Include Files

- `mkl.fi`

Description

The routine `?ptts2` solves a tridiagonal system of the form

$$A*X = B$$

Real flavors `sptts2/dptts2` use the $L*D*L^T$ factorization of *A* computed by `spttrf/dpttrf`, and complex flavors `cptts2/zptts2` use the U^H*D*U or $L*D*L^H$ factorization of *A* computed by `cpttrf/zpttrf`.

D is a diagonal matrix specified in the vector *d*, *U* (or *L*) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector *e*, and *X* and *B* are *n*-by-*nrhs* matrices.

Input Parameters

<i>iuplo</i>	<p>INTEGER. Used with complex flavors only.</p> <p>Specifies the form of the factorization, and whether the vector <i>e</i> is the superdiagonal of the upper bidiagonal factor <i>U</i> or the subdiagonal of the lower bidiagonal factor <i>L</i>.</p> <p>= 1: $A = U^H * D * U$, <i>e</i> is the superdiagonal of <i>U</i>; = 0: $A = L * D * L^H$, <i>e</i> is the subdiagonal of <i>L</i></p>
<i>n</i>	<p>INTEGER. The order of the tridiagonal matrix <i>A</i>. $n \geq 0$.</p>
<i>nrhs</i>	<p>INTEGER. The number of right hand sides, that is, the number of columns of the matrix <i>B</i>. $nrhs \geq 0$.</p>
<i>d</i>	<p>REAL for <i>sptts2/cptts2</i> DOUBLE PRECISION for <i>dptts2/zptts2</i>.</p> <p>Array, DIMENSION (<i>n</i>).</p> <p>The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization of <i>A</i>.</p>
<i>e</i>	<p>REAL for <i>sptts2</i> DOUBLE PRECISION for <i>dptts2</i> COMPLEX for <i>cptts2</i> DOUBLE COMPLEX for <i>zptts2</i>.</p> <p>Array, DIMENSION (<i>n</i>-1).</p> <p>Contains the (<i>n</i>-1) subdiagonal elements of the unit bidiagonal factor <i>L</i> from the $L * D * L^T$ (for real flavors) or $L * D * L^H$ (for complex flavors when <i>iuplo</i> = 0) factorization of <i>A</i>.</p> <p>For complex flavors when <i>iuplo</i> = 1, <i>e</i> contains the (<i>n</i>-1) superdiagonal elements of the unit bidiagonal factor <i>U</i> from the factorization $A = U^H * D * U$.</p>
<i>B</i>	<p>REAL for <i>sptts2/cptts2</i> DOUBLE PRECISION for <i>dptts2/zptts2</i>.</p> <p>Array, DIMENSION (<i>ldb</i>, <i>nrhs</i>).</p> <p>On entry, the right hand side vectors <i>B</i> for the system of linear equations.</p>
<i>ldb</i>	<p>INTEGER. The leading dimension of the array <i>B</i>. $ldb \geq \max(1, n)$.</p>

Output Parameters

<i>b</i>	On exit, the solution vectors, <i>X</i> .
----------	---

?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call srscl( n, sa, sx, incx )
call drscl( n, sa, sx, incx )
call csrscl( n, sa, sx, incx )
call zdrscl( n, sa, sx, incx )
```

Include Files

- mkl.fi

Description

The routine `?rscl` multiplies an n -element real/complex vector x by the real scalar $1/a$. This is done without overflow or underflow as long as the final result x/a does not overflow or underflow.

Input Parameters

n	INTEGER. The number of components of the vector x .
sa	REAL for <code>srscl/csrscl</code> DOUBLE PRECISION for <code>drscl/zdrscl</code> . The scalar a which is used to divide each component of the vector x . sa must be ≥ 0 , or the subroutine will divide by zero.
sx	REAL for <code>srscl</code> DOUBLE PRECISION for <code>drscl</code> COMPLEX for <code>csrscl</code> DOUBLE COMPLEX for <code>zdrscl</code> . Array, DIMENSION $(1+(n-1)* incx)$. The n -element vector x .
$incx$	INTEGER. The increment between successive values of the vector sx . If $incx > 0$, $sx(1)=x(1)$, and $sx(1+(i-1)*incx)=x(i)$, $1 < i \leq n$.

Output Parameters

sx	On exit, the result x/a .
------	-----------------------------

?syswapr

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```
call ssyswapr( uplo, n, a, lda, i1, i2 )
call dsyswapr( uplo, n, a, lda, i1, i2 )
call csyswapr( uplo, n, a, lda, i1, i2 )
call zsyswapr( uplo, n, a, lda, i1, i2 )
```

```
call syswapr( a, i1, i2[, uplo] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L^*D^*L^T$.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	REAL for <i>ssyswapr</i> DOUBLE PRECISION for <i>dsyswapr</i> COMPLEX for <i>csyswapr</i> DOUBLE COMPLEX for <i>zsyswapr</i> Array of size (<i>lda</i> , <i>n</i>). The array <i>a</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <i>?sytrf</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>i1</i>	INTEGER. Index of the first row to swap.
<i>i2</i>	INTEGER. Index of the second row to swap.

Output Parameters

<i>a</i>	If <i>info</i> = 0, the symmetric inverse of the original matrix. If <i>info</i> = 'U', the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced. If <i>info</i> = 'L', the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.
----------	---

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *syswapr* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>i1</i>	Holds the index for swap.
<i>i2</i>	Holds the index for swap.
<i>uplo</i>	Indicates how the matrix <i>A</i> has been factored. Must be 'U' or 'L'.

See Also

?sytrf

?heswapr

Applies an elementary permutation on the rows and columns of a Hermitian matrix.

Syntax

```
call cheswapr( uplo, n, a, lda, i1, i2 )
call zheswapr( uplo, n, a, lda, i1, i2 )
call heswapr( a, i1, i2 [,uplo] )
```

Include Files

- mkl.fi, lapack.f90

Description

The routine applies an elementary permutation on the rows and columns of a Hermitian matrix.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	COMPLEX for cheswapr DOUBLE COMPLEX for zheswapr Array of size (<i>lda</i> , <i>n</i>). The array <i>a</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by ?hetrf.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,n)$.

i1 INTEGER. Index of the first row to swap.

i2 INTEGER. Index of the second row to swap.

Output Parameters

a If *info* = 0, the inverse of the original matrix.

If *info* = 'U', the upper triangular part of the inverse is formed and the part of *A* below the diagonal is not referenced.

If *info* = 'L', the lower triangular part of the inverse is formed and the part of *A* above the diagonal is not referenced.

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine `heswapr` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

i1 Holds the index for swap.

i2 Holds the index for swap.

uplo Must be 'U' or 'L'.

See Also

[?hetrf](#)
[al?syswapr1](#)

[al?syswapr1](#)

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```
call ssyswapr1( uplo, n, a, lda, i1, i2 )
call dsyswapr1( uplo, n, a, lda, i1, i2 )
call csyswapr1( uplo, n, a, lda, i1, i2 )
call zsyswapr1( uplo, n, a, lda, i1, i2 )
call syswapr1( a, i1, i2[, uplo] )
```

Include Files

- `mk1.fi`, `lapack.f90`

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Indicates how the input matrix <i>A</i> has been factored:</p> <p>If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U^*D^*U^T$.</p> <p>If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L^*D^*L^T$.</p>
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	<p>REAL for <i>ssyswapr1</i></p> <p>DOUBLE PRECISION for <i>dsyswapr1</i></p> <p>COMPLEX for <i>csyswapr1</i></p> <p>DOUBLE COMPLEX for <i>zsyswapr1</i></p> <p>Array of dimension (lda, n).</p> <p>The array <i>a</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <i>?sytrf</i>.</p>
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>i1</i>	INTEGER. Index of the first row to swap.
<i>i2</i>	INTEGER. Index of the second row to swap.

Output Parameters

<i>a</i>	<p>If <i>info</i> = 0, the symmetric inverse of the original matrix.</p> <p>If <i>info</i> = 'U', the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced.</p> <p>If <i>info</i> = 'L', the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.</p>
----------	--

LAPACK 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [LAPACK 95 Interface Conventions](#).

Specific details for the routine *syswapr1* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>i1</i>	Holds the index for swap.
<i>i2</i>	Holds the index for swap.
<i>uplo</i>	Indicates how the matrix <i>A</i> has been factored. Must be 'U' or 'L'.

See Also`?sytrf`**?sygs2/?hegs2**

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from `?potrf` (unblocked algorithm).

Syntax

```
call ssysgs2( itype, uplo, n, a, lda, b, ldb, info )
```

```
call dsysgs2( itype, uplo, n, a, lda, b, ldb, info )
```

```
call csysgs2( itype, uplo, n, a, lda, b, ldb, info )
```

```
call zsysgs2( itype, uplo, n, a, lda, b, ldb, info )
```

Include Files

- `mkl.fi`

Description

The routine `?sygs2/?hegs2` reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

If `itype = 1`, the problem is

$$A*x = \lambda*B*x$$

and `A` is overwritten by $\text{inv}(U^H)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^H)$ for complex flavors and by $\text{inv}(U^T)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^T)$ for real flavors.

If `itype = 2` or `3`, the problem is

$$A*B*x = \lambda*x, \text{ or } B*A*x = \lambda*x,$$

and `A` is overwritten by $U*A*U^H$ or L^H*A*L for complex flavors and by $U*A*U^T$ or L^T*A*L for real flavors. Here U^T and L^T are the transpose while U^H and L^H are conjugate transpose of `U` and `L`.

`B` must be previously factorized by `?potrf` as follows:

- U^H*U or $L*L^H$ for complex flavors
- U^T*U or $L*L^T$ for real flavors

Input Parameters`itype`

INTEGER.

For complex flavors:

= 1: compute $\text{inv}(U^H)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^H)$;= 2 or 3: compute $U*A*U^H$ or L^H*A*L .

For real flavors:

= 1: compute $\text{inv}(U^T)*A*\text{inv}(U)$ or $\text{inv}(L)*A*\text{inv}(L^T)$;= 2 or 3: compute $U*A*U^T$ or L^T*A*L .`uplo`

CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix `A` is stored, and how `B` has been factorized.

	= 'U': upper triangular
	= 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> . $n \geq 0$.
<i>a</i>	REAL for <i>ssygs2</i> DOUBLE PRECISION for <i>dsygs2</i> COMPLEX for <i>chegs2</i> DOUBLE COMPLEX for <i>zhegs2</i> . Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the symmetric/Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>b</i>	REAL for <i>ssygs2</i> DOUBLE PRECISION for <i>dsygs2</i> COMPLEX for <i>chegs2</i> DOUBLE COMPLEX for <i>zhegs2</i> . Array, DIMENSION (<i>ldb</i> , <i>n</i>). The triangular factor from the Cholesky factorization of <i>B</i> as returned by ? <i>potrf</i> .
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . $ldb \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, If <i>info</i> = 0, the transformed matrix, stored in the same format as <i>A</i> .
<i>info</i>	INTEGER. = 0: successful exit. < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation(unblocked algorithm).

Syntax

```
call ssytd2( uplo, n, a, lda, d, e, tau, info )
call dsytd2( uplo, n, a, lda, d, e, tau, info )
call chetd2( uplo, n, a, lda, d, e, tau, info )
call zhetd2( uplo, n, a, lda, d, e, tau, info )
```

Include Files

- mkl.fi

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix A to real symmetric tridiagonal form T by an orthogonal/unitary similarity transformation: $Q^T A Q = T$ ($Q^H A Q = T$).

Input Parameters

uplo CHARACTER*1.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored:
= 'U': upper triangular
= 'L': lower triangular

n INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for ssytd2
DOUBLE PRECISION for dsytd2
COMPLEX for chetd2
DOUBLE COMPLEX for zhetd2.
Array, DIMENSION (*lda*, *n*).
On entry, the symmetric/Hermitian matrix A .
If *uplo* = 'U', the leading n -by- n upper triangular part of *a* contains the upper triangular part of the matrix A , and the strictly lower triangular part of *a* is not referenced.
If *uplo* = 'L', the leading n -by- n lower triangular part of *a* contains the lower triangular part of the matrix A , and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *a* are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors;

if $uplo = 'L'$, the diagonal and first subdiagonal of a are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

d

REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION (n).

The diagonal elements of the tridiagonal matrix T :

$$d(i) = a(i, i).$$

e

REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION ($n-1$).

The off-diagonal elements of the tridiagonal matrix T :

$$e(i) = a(i, i+1) \text{ if } uplo = 'U',$$

$$e(i) = a(i+1, i) \text{ if } uplo = 'L'.$$

tau

REAL for ssytd2
 DOUBLE PRECISION for dsytd2
 COMPLEX for chetd2
 DOUBLE COMPLEX for zhetd2.

Array, DIMENSION (n).

The first $n-1$ elements contain scalar factors of the elementary reflectors. $tau(n)$ is used as workspace.

$info$

INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2( uplo, n, a, lda, ipiv, info )
call dsytf2( uplo, n, a, lda, ipiv, info )
call csytf2( uplo, n, a, lda, ipiv, info )
call zsytf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine `?sytff2` computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

$$A = U^*D^*U^T, \text{ or } A = L^*D^*L^T,$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored = 'U': upper triangular = 'L': lower triangular
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$.
<code>a</code>	REAL for <code>ssytf2</code> DOUBLE PRECISION for <code>dsytf2</code> COMPLEX for <code>csytf2</code> DOUBLE COMPLEX for <code>zsytf2</code> . Array, DIMENSION (lda, n). On entry, the symmetric matrix A . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.
<code>lda</code>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<code>a</code>	On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .
<code>ipiv</code>	INTEGER. Array, DIMENSION (n). Details of the interchanges and the block structure of D If <code>ipiv(k) > 0</code> , then rows and columns k and <code>ipiv(k)</code> are interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ are interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument has an illegal value

> 0: if $info = k$, $D(k,k)$ is exactly zero. The factorization are completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sytf2_rook

Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2_rook( uplo, n, a, lda, ipiv, info )
```

```
call dsytf2_rook( uplo, n, a, lda, ipiv, info )
```

```
call csytf2_rook( uplo, n, a, lda, ipiv, info )
```

```
call zsytf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine `?sytf2_rook` computes the factorization of a real/complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

$$A = U^*D^*U^T, \text{ or } A = L^*D^*L^T,$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored

= 'U': upper triangular

= 'L': lower triangular

n

INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for `ssytf2_rook`
 DOUBLE PRECISION for `dsytf2_rook`
 COMPLEX for `csytf2_rook`
 DOUBLE COMPLEX for `zsytf2_rook`.
 Array, DIMENSION (*lda*, *n*).
 On entry, the symmetric matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER.
 The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv INTEGER.
 Array, DIMENSION (*n*).
 Details of the interchanges and the block structure of *D*
 If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.
 If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k* - 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* - 1 and -*ipiv*(*k* - 1) were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.
 If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k* + 1) < 0, then rows and columns *k* and -*ipiv*(*k*) were interchanged, rows and columns *k* + 1 and -*ipiv*(*k* + 1) were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*k*, the *k*-th argument has an illegal value
 > 0: if *info* = *k*, $D(k,k)$ is exactly zero. The factorization are completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2( uplo, n, a, lda, ipiv, info )
call zhetf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

$$A = U^*D^*U^H \text{ or } A = L^*D^*L^H$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of U , and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>A</i>	COMPLEX for chetf2 DOUBLE COMPLEX for zhetf2. Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .
<i>ipiv</i>	INTEGER. Array, DIMENSION (n). Details of the interchanges and the block structure of D If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument had an illegal value

> 0: if $info = k$, $D(k,k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

zhetf2_rook

Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call zhetf2_rook( uplo, n, a, lda, ipiv, info )
```

```
call zhetf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- mkl.fi

Description

The routine computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

$$A = U^* D^* U^H \text{ or } A = L^* D^* L^H$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of U , and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n

INTEGER. The order of the matrix A . $n \geq 0$.

a

COMPLEX for `zhetf2_rook`

DOUBLE COMPLEX for `zhetf2_rook`.

Array, DIMENSION (lda, n).

On entry, the Hermitian matrix A .

If $uplo = 'U'$, the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .

$ipiv$ INTEGER. Array, DIMENSION (n).

Details of the interchanges and the block structure of D .

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

$info$ INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument had an illegal value

> 0: if $info = k$, $D(k,k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work,
            lwork, info )
```

```
call dtgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work,
            lwork, info )
```

```
call ctgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

```
call ztgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Include Files

- `mkl.fi`

Description

The real routines `stgex2/dtgex2` swap adjacent diagonal blocks (A_{11} , B_{11}) and (A_{22} , B_{22}) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair (A, B) by an orthogonal equivalence transformation. (A, B) must be in generalized real Schur canonical form (as returned by `sgges/dgges`), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex routines `ctgex2/ztgex2` swap adjacent diagonal 1-by-1 blocks (A_{11} , B_{11}) and (A_{22} , B_{22}) in an upper triangular matrix pair (A, B) by a unitary equivalence transformation.

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

All routines optionally update the matrices Q and Z of generalized Schur vectors:

For real flavors,

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})^T = Q(\text{out}) * A(\text{out}) * Z(\text{out})^T$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})^T = Q(\text{out}) * B(\text{out}) * Z(\text{out})^T.$$

For complex flavors,

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})^H = Q(\text{out}) * A(\text{out}) * Z(\text{out})^H$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})^H = Q(\text{out}) * B(\text{out}) * Z(\text{out})^H.$$

Input Parameters

<code>wantq</code>	LOGICAL. If <code>wantq = .TRUE.</code> : update the left transformation matrix Q ; If <code>wantq = .FALSE.</code> : do not update Q .
<code>wantz</code>	LOGICAL. If <code>wantz = .TRUE.</code> : update the right transformation matrix Z ; If <code>wantz = .FALSE.</code> : do not update Z .
<code>n</code>	INTEGER. The order of the matrices A and B . $n \geq 0$.
<code>a, b</code>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> COMPLEX for <code>ctgex2</code> DOUBLE COMPLEX for <code>ztgex2</code> . Arrays, DIMENSION (lda, n) and (ldb, n) , respectively. On entry, the matrices A and B in the pair (A, B) .
<code>lda</code>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of the array b . $ldb \geq \max(1, n)$.
<code>q, z</code>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> COMPLEX for <code>ctgex2</code> DOUBLE COMPLEX for <code>ztgex2</code> . Arrays, DIMENSION (ldq, n) and (ldz, n) , respectively.

	On entry, if <code>wantq = .TRUE.</code> , <code>q</code> contains the orthogonal/unitary matrix <code>Q</code> , and if <code>wantz = .TRUE.</code> , <code>z</code> contains the orthogonal/unitary matrix <code>Z</code> .
<code>ldq</code>	INTEGER. The leading dimension of the array <code>q</code> . $ldq \geq 1$. If <code>wantq = .TRUE.</code> , $ldq \geq n$.
<code>ldz</code>	INTEGER. The leading dimension of the array <code>z</code> . $ldz \geq 1$. If <code>wantz = .TRUE.</code> , $ldz \geq n$.
<code>j1</code>	INTEGER. The index to the first block (<code>A11</code> , <code>B11</code>). $1 \leq j1 \leq n$.
<code>n1</code>	INTEGER. Used with real flavors only. The order of the first block (<code>A11</code> , <code>B11</code>). $n1 = 0, 1$ or 2 .
<code>n2</code>	INTEGER. Used with real flavors only. The order of the second block (<code>A22</code> , <code>B22</code>). $n2 = 0, 1$ or 2 .
<code>work</code>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> . Workspace array, DIMENSION ($\max(1, lwork)$). Used with real flavors only.
<code>lwork</code>	INTEGER. The dimension of the array <code>work</code> . $lwork \geq \max(n * (n2 + n1), 2 * (n2 + n1)^2)$

Output Parameters

<code>a</code>	On exit, the updated matrix <code>A</code> .
<code>B</code>	On exit, the updated matrix <code>B</code> .
<code>Q</code>	On exit, the updated matrix <code>Q</code> . Not referenced if <code>wantq = .FALSE.</code>
<code>z</code>	On exit, the updated matrix <code>Z</code> . Not referenced if <code>wantz = .FALSE.</code>
<code>info</code>	INTEGER. =0: Successful exit For <code>stgex2/dtgex2</code> : If <code>info = 1</code> , the transformed matrix (<code>A</code> , <code>B</code>) would be too far from generalized Schur form; the blocks are not swapped and (<code>A</code> , <code>B</code>) and (<code>Q</code> , <code>Z</code>) are unchanged. The problem of swapping is too ill-conditioned. If <code>info = -16</code> : <code>lwork</code> is too small. Appropriate value for <code>lwork</code> is returned in <code>work(1)</code> . For <code>ctgex2/ztgex2</code> : If <code>info = 1</code> , the transformed matrix pair (<code>A</code> , <code>B</code>) would be too far from generalized Schur form; the problem is ill-conditioned.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

```
call stgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

```
call dtgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

```
call ctgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

```
call ztgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

Include Files

- mkl.fi

Description

The routine ?tgsy2 solves the generalized Sylvester equation:

$$A^*R - L^*B = \text{scale} * C \quad (1)$$

$$D^*R - L^*E = \text{scale} * F$$

using Level 1 and 2 BLAS, where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively. For *stgsy2/dtgsy2*, pairs (A, D) and (B, E) must be in generalized Schur canonical form, that is, A, B are upper quasi triangular and D, E are upper triangular. For *ctgsy2/ztgsy2*, matrices A, B, D and E are upper triangular (that is, (A, D) and (B, E) in generalized Schur form).

The solution (R, L) overwrites (C, F) .

$0 \leq \text{scale} \leq 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

$$Z * x = \text{scale} * b$$

where Z is defined for real flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^T, I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E^T, I_m) \end{bmatrix} \quad (2)$$

and for complex flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^H, I_m) \\ \text{kron}(I_n, D) & -\text{kron}(E^H, I_m) \end{bmatrix} \quad (3)$$

Here I_k is the identity matrix of size k and X^T (X^H) is the transpose (conjugate transpose) of X . $\text{kron}(X, Y)$ denotes the Kronecker product between the matrices X and Y .

For real flavors, if $trans = 'T'$, solve the transposed system

$$Z^T * y = scale * b$$

for y , which is equivalent to solving for R and L in

$$A^T * R + D^T * L = scale * C \quad (4)$$

$$R * B^T + L * E^T = scale * (-F)$$

For complex flavors, if $trans = 'C'$, solve the conjugate transposed system

$$Z^H * y = scale * b$$

for y , which is equivalent to solving for R and L in

$$A^H * R + D^H * L = scale * C \quad (5)$$

$$R * B^H + L * E^H = scale * (-F)$$

These cases are used to compute an estimate of $\text{Dif}[(A, D), (B, E)] = \text{sigma_min}(Z)$ using reverse communication with [?lacon](#).

[?tgsy2](#) also (for $ijob \geq 1$) contributes to the computation in [?tgsy1](#) of an upper bound on the separation between two matrix pairs. Then the input $(A, D), (B, E)$ are sub-pencils of the matrix pair (two matrix pairs) in [?tgsy1](#). See [?tgsy1](#) for details.

Input Parameters

<i>trans</i>	CHARACTER*1. If $trans = 'N'$, solve the generalized Sylvester equation (1); If $trans = 'T'$: solve the transposed system (4). If $trans = 'C'$: solve the conjugate transposed system (5).
<i>ijob</i>	INTEGER. Specifies what kind of functionality is to be performed. If $ijob = 0$: solve (1) only. If $ijob = 1$: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used); If $ijob = 2$: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used). Not referenced if $trans = 'T'$.
<i>m</i>	INTEGER. On entry, m specifies the order of A and D , and the row dimension of C, F, R and L .
<i>n</i>	INTEGER. On entry, n specifies the order of B and E , and the column dimension of C, F, R and L .
<i>a, b</i>	REAL for stgsy2 DOUBLE PRECISION for dtgsy2 COMPLEX for ctgsy2 DOUBLE COMPLEX for ztgsy2 .

Arrays, DIMENSION (lda, m) and (ldb, n) , respectively. On entry, a contains an upper (quasi) triangular matrix A , and b contains an upper (quasi) triangular matrix B .

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.

ldb INTEGER.

The leading dimension of the array b . $ldb \geq \max(1, n)$.

c, f REAL for stgtsy2
DOUBLE PRECISION for dtgtsy2
COMPLEX for ctgtsy2
DOUBLE COMPLEX for ztgtsy2.

Arrays, DIMENSION (ldc, n) and (ldf, n) , respectively. On entry, c contains the right-hand-side of the first matrix equation in (1), and f contains the right-hand-side of the second matrix equation in (1).

ldc INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.

d, e REAL for stgtsy2
DOUBLE PRECISION for dtgtsy2
COMPLEX for ctgtsy2
DOUBLE COMPLEX for ztgtsy2.

Arrays, DIMENSION (ldd, m) and (lde, n) , respectively. On entry, d contains an upper triangular matrix D , and e contains an upper triangular matrix E .

ldd INTEGER. The leading dimension of the array d . $ldd \geq \max(1, m)$.

lde INTEGER. The leading dimension of the array e . $lde \geq \max(1, n)$.

ldf INTEGER. The leading dimension of the array f . $ldf \geq \max(1, m)$.

$rdsum$ REAL for stgtsy2/ctgtsy2
DOUBLE PRECISION for dtgtsy2/ztgtsy2.

On entry, the sum of squares of computed contributions to the Dif-estimate under computation by ?tg_{sy}L, where the scaling factor $rdscal$ has been factored out.

$rdscal$ REAL for stgtsy2/ctgtsy2
DOUBLE PRECISION for dtgtsy2/ztgtsy2.

On entry, scaling factor used to prevent overflow in $rdsum$.

$iwork$ INTEGER. Used with real flavors only.

Workspace array, DIMENSION $(m+n+2)$.

Output Parameters

c On exit, if $ijob = 0$, c is overwritten by the solution R .

<i>f</i>	On exit, if <i>ijob</i> = 0, <i>f</i> is overwritten by the solution <i>L</i> .
<i>scale</i>	REAL for <i>stgsy2/ctgsy2</i> DOUBLE PRECISION for <i>dtgsy2/ztgsy2</i> . On exit, $0 \leq scale \leq 1$. If $0 < scale < 1$, the solutions <i>R</i> and <i>L</i> (<i>C</i> and <i>F</i> on entry) hold the solutions to a slightly perturbed system, but the input matrices <i>A</i> , <i>B</i> , <i>D</i> and <i>E</i> are not changed. If <i>scale</i> = 0, <i>R</i> and <i>L</i> hold the solutions to the homogeneous system with <i>C</i> = <i>F</i> = 0. Normally <i>scale</i> = 1.
<i>rdsum</i>	On exit, the corresponding sum of squares updated with the contributions from the current sub-system. If <i>trans</i> = 'T', <i>rdsum</i> is not touched. Note that <i>rdsum</i> only makes sense when ?tgsy2 is called by ?tgsy1.
<i>rdscal</i>	On exit, <i>rdscal</i> is updated with respect to the current contributions in <i>rdsum</i> . If <i>trans</i> = 'T', <i>rdscal</i> is not touched. Note that <i>rdscal</i> only makes sense when ?tgsy2 is called by ?tgsy1.
<i>pq</i>	INTEGER. Used with real flavors only. On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine <i>stgsy2/dtgsy2</i> .
<i>info</i>	INTEGER. On exit, if <i>info</i> is set to = 0: Successful exit < 0: If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument has an illegal value. > 0: The matrix pairs (<i>A</i> , <i>D</i>) and (<i>B</i> , <i>E</i>) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix (unblocked algorithm).

Syntax

```
call strti2( uplo, diag, n, a, lda, info )
call dtrti2( uplo, diag, n, a, lda, info )
call ctrti2( uplo, diag, n, a, lda, info )
call ztrti2( uplo, diag, n, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine ?trti2 computes the inverse of a real/complex upper or lower triangular matrix.

This is the *Level 2 BLAS* version of the algorithm.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular. = 'U': upper triangular = 'L': lower triangular
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix <i>A</i> is unit triangular. = 'N': non-unit triangular = 'N': non-unit triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	REAL for <i>strti2</i> DOUBLE PRECISION for <i>dtrti2</i> COMPLEX for <i>ctrti2</i> DOUBLE COMPLEX for <i>ztrti2</i> . Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the triangular matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>a</i> is not referenced. If <i>diag</i> = 'U', the diagonal elements of <i>a</i> are also not referenced and are assumed to be 1.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, the (triangular) inverse of the original matrix, in the same storage format.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th argument had an illegal value

clag2z

Converts a complex single precision matrix to a complex double precision matrix.

Syntax

```
call clag2z( m, n, sa, ldsa, a, lda, info )
```

Include Files

- `mkl.fi`

Description

This routine converts a complex single precision matrix SA to a complex double precision matrix A .

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix A ($n \geq 0$).
<i>lds</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $lds \geq \max(1, m)$.
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the m -by- n coefficient matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>lds</i> , <i>n</i>). On exit, contains the m -by- n coefficient matrix SA .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

dlag2s

Converts a double precision matrix to a single precision matrix.

Syntax

```
call dlag2s( m, n, a, lda, sa, lds, info )
```

Include Files

- `mkl.fi`

Description

This routine converts a double precision matrix SA to a single precision matrix A .

`RMAX` is the overflow for the single precision arithmetic. `dlag2s` checks that all the entries of A are between $-RMAX$ and $RMAX$. If not, the conversion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, m)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; $ldsa \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, if <i>info</i> = 0, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> ; if <i>info</i> > 0, the content of <i>sa</i> is unspecified.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = 1, an entry of the matrix <i>A</i> is greater than the single precision overflow threshold; in this case, the content of <i>sa</i> on exit is unspecified.

slag2d

Converts a single precision matrix to a double precision matrix.

Syntax

```
call slag2d( m, n, sa, ldsa, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine converts a single precision matrix *SA* to a double precision matrix *A*.

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix <i>A</i> ($n \geq 0$).
<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On entry, contains the <i>m</i> -by- <i>n</i> coefficient matrix <i>SA</i> .

ldsa INTEGER. The leading dimension of the array *sa*; $ldsa \geq \max(1, m)$.

lda INTEGER. The leading dimension of the array *a*; $lda \geq \max(1, m)$.

Output Parameters

a DOUBLE PRECISION array, DIMENSION (*lda*, *n*).
On exit, contains the *m*-by-*n* coefficient matrix *A*.

info INTEGER.
If *info* = 0, the execution is successful.

zlag2c

Converts a complex double precision matrix to a complex single precision matrix.

Syntax

```
call zlag2c( m, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

The routine converts a double precision complex matrix *SA* to a single precision complex matrix *A*.

RMAX is the overflow for the single precision arithmetic. `zlag2c` checks that all the entries of *A* are between -RMAX and RMAX. If not, the conversion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m INTEGER. The number of lines of the matrix *A* ($m \geq 0$).

n INTEGER. The number of columns in the matrix *A* ($n \geq 0$).

a DOUBLE COMPLEX array, DIMENSION (*lda*, *n*).
On entry, contains the *m*-by-*n* coefficient matrix *A*.

lda INTEGER. The leading dimension of the array *a*; $lda \geq \max(1, m)$.

ldsa INTEGER. The leading dimension of the array *sa*; $ldsa \geq \max(1, m)$.

Output Parameters

sa COMPLEX array, DIMENSION (*ldsa*, *n*).
On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*; if *info* > 0, the content of *sa* is unspecified.

info INTEGER.
If *info* = 0, the execution is successful.

If $info = 1$, an entry of the matrix A is greater than the single precision overflow threshold; in this case, the content of sa on exit is unspecified.

?larfp

Generates a real or complex elementary reflector.

Syntax

```
call slarfp(n, alpha, x, incx, tau)
call dlarfp(n, alpha, x, incx, tau)
call clarfp(n, alpha, x, incx, tau)
call zlarfp(n, alpha, x, incx, tau)
```

Include Files

- mkl.fi

Description

The ?larfp routines generate a real or complex elementary reflector H of order n , such that

$$H \begin{pmatrix} \alpha \\ x \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \end{pmatrix},$$

and $H^*H = I$ for real flavors, $\text{conjg}(H)^*H = I$ for complex flavors.

Here

α and β are scalars, β is real and non-negative,

x is $(n-1)$ -element vector.

H is represented in the form

$$H = I - \tau \begin{pmatrix} 1 & \\ & v \end{pmatrix} \begin{pmatrix} 1 & v' \\ & v \end{pmatrix},$$

where τ is scalar, and v is $(n-1)$ -element vector .

For real flavors if the elements of x are all zero, then $\tau = 0$ and H is taken to be the unit matrix. Otherwise $1 \leq \tau \leq 2$.

For complex flavors if the elements of x are all zero and α is real, then $\tau = 0$ and H is taken to be the unit matrix. Otherwise $1 \leq \text{real}(\tau) \leq 2$, and $\text{abs}(\tau - 1) \leq 1$.

Input Parameters

n	INTEGER. Specifies the order of the elementary reflector.
α	REAL for slarfp DOUBLE PRECISION for dlarfp COMPLEX for clarfp DOUBLE COMPLEX for zlarfp Specifies the scalar α .
x	REAL for slarfp

DOUBLE PRECISION for dlarfp

COMPLEX for clarfp

DOUBLE COMPLEX for zlarfp

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$. It contains the vector x .

incx INTEGER. Specifies the increment for the elements of x .
The value of *incx* must not be zero.

Output Parameters

alpha Overwritten by the value *beta*.

y Overwritten by the vector v .

tau REAL for slarfp
DOUBLE PRECISION for dlarfp
COMPLEX for clarfp
DOUBLE COMPLEX for zlarfp
Contains the scalar *tau*.

ila?lc

Scans a matrix for its last non-zero column.

Syntax

value = ilaslc(*m*, *n*, *a*, *lda*)

value = iladlc(*m*, *n*, *a*, *lda*)

value = ilaclc(*m*, *n*, *a*, *lda*)

value = ilazlc(*m*, *n*, *a*, *lda*)

Include Files

- mkl.fi

Description

The *ila?lc* routines scan a matrix A for its last non-zero column.

Input Parameters

m INTEGER. Specifies number of rows in the matrix A .

n INTEGER. Specifies number of columns in the matrix A .

a REAL for ilaslc
DOUBLE PRECISION for iladlc
COMPLEX for ilaclc

DOUBLE COMPLEX for `ilazlc`

Array, DIMENSION(`lda`, *). The second dimension of `a` must be at least $\max(1, n)$.

Before entry the leading n -by- n part of the array `a` must contain the matrix `A`.

`lda` INTEGER. Specifies the leading dimension of `a` as declared in the calling (sub)program. The value of `lda` must be at least $\max(1, m)$.

Output Parameters

`value` INTEGER
Number of the last non-zero column.

ila?lr

Scans a matrix for its last non-zero row.

Syntax

`value` = `ilaslr`(`m`, `n`, `a`, `lda`)

`value` = `iladlr`(`m`, `n`, `a`, `lda`)

`value` = `ilaclr`(`m`, `n`, `a`, `lda`)

`value` = `ilazlr`(`m`, `n`, `a`, `lda`)

Include Files

- `mkl.fi`

Description

The `ila?lr` routines scan a matrix `A` for its last non-zero row.

Input Parameters

`m` INTEGER. Specifies number of rows in the matrix `A`.

`n` INTEGER. Specifies number of columns in the matrix `A`.

`a` REAL for `ilaslr`

DOUBLE PRECISION for `iladlr`

COMPLEX for `ilaclr`

DOUBLE COMPLEX for `idazlr`

Array, DIMENSION(`lda`, *). The second dimension of `a` must be at least $\max(1, n)$.

Before entry the leading n -by- n part of the array `a` must contain the matrix `A`.

`lda` INTEGER. Specifies the leading dimension of `a` as declared in the calling (sub)program. The value of `lda` must be at least $\max(1, m)$.

Output Parameters

value INTEGER
Number of the last non-zero row.

?gsvj0

Pre-processor for the routine ?gesvj.

Syntax

call sgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

call dgsvj0(*jobv*, *m*, *n*, *a*, *lda*, *d*, *sva*, *mv*, *v*, *ldv*, *eps*, *sfmin*, *tol*, *nsweep*, *work*, *lwork*, *info*)

Include Files

- `mk1.fi`

Description

This routine is called from ?gesvj as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as ?gesvj does, but it does not check convergence (stopping criterion).

The routine ?gsvj0 enables ?gesvj to use a simplified version of itself to work on a submatrix of the original matrix.

Input Parameters

jobv CHARACTER*1. Must be 'V', 'A', or 'N'.
Specifies whether the output from this routine is used to compute the matrix *V*.
If *jobv* = 'V', the product of the Jacobi rotations is accumulated by post-multiplying the *n*-by-*n* array *v*.
If *jobv* = 'A', the product of the Jacobi rotations is accumulated by post-multiplying the *mv*-by-*n* array *v*.
If *jobv* = 'N', the Jacobi rotations are not accumulated.

m INTEGER. The number of rows of the input matrix *A* ($m \geq 0$).

n INTEGER. The number of columns of the input matrix *B* ($m \geq n \geq 0$).

a REAL for sgsvj0
DOUBLE PRECISION for dgsvj0.
Arrays, DIMENSION (*lda*, *). Contains the *m*-by-*n* matrix *A*, such that $A \cdot \text{diag}(D)$ represents the input matrix. The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

d REAL for sgsvj0

	DOUBLE PRECISION for <code>dgsvj0</code> .
	Arrays , DIMENSION (n). Contains the diagonal matrix D that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry $A*\text{diag}(D)$ represents the input matrix.
<code>sva</code>	REAL for <code>sgsvj0</code> DOUBLE PRECISION for <code>dgsvj0</code> .
	Arrays , DIMENSION (n). Contains the Euclidean norms of the columns of the matrix $A*\text{diag}(D)$.
<code>mv</code>	INTEGER. The leading dimension of b ; at least $\max(1, p)$. If <code>jobv = 'A'</code> , then mv rows of v are post-multiplied by a sequence of Jacobi rotations. If <code>jobv = 'N'</code> , then mv is not referenced.
<code>v</code>	REAL for <code>sgsvj0</code> DOUBLE PRECISION for <code>dgsvj0</code> . Array , DIMENSION ($ldv, *$). The second dimension of a must be at least $\max(1, n)$. If <code>jobv = 'V'</code> , then n rows of v are post-multiplied by a sequence of Jacobi rotations. If <code>jobv = 'A'</code> , then mv rows of v are post-multiplied by a sequence of Jacobi rotations. If <code>jobv = 'N'</code> , then v is not referenced.
<code>ldv</code>	INTEGER. The leading dimension of the array v ; $ldv \geq 1$ $ldv \geq n$ if <code>jobv = 'V'</code> ; $ldv \geq mv$ if <code>jobv = 'A'</code> .
<code>eps</code>	REAL for <code>sgsvj0</code> DOUBLE PRECISION for <code>dgsvj0</code> . The relative machine precision (epsilon) returned by the routine ?lamch .
<code>sfmin</code>	REAL for <code>sgsvj0</code> DOUBLE PRECISION for <code>dgsvj0</code> . Value of safe minimum returned by the routine ?lamch .
<code>tol</code>	REAL for <code>sgsvj0</code> DOUBLE PRECISION for <code>dgsvj0</code> . The threshold for Jacobi rotations. For a pair $A(:, p), A(:, q)$ of pivot columns, the Jacobi rotation is applied only if $\text{abs}(\cos(\text{angle}(A(:, p), A(:, q)))) > \text{tol}$.
<code>nsweep</code>	INTEGER. The number of sweeps of Jacobi rotations to be performed.

work REAL for `sgsvj0`
 DOUBLE PRECISION for `dgsvj0`.
 Workspace array, DIMENSION (*lwork*).

lwork INTEGER. The size of the array *work*; at least $\max(1, m)$.

Output Parameters

a On exit, $A \cdot \text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively

d On exit, $A \cdot \text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in *tol* and *nsweep*, respectively.

sva On exit, contains the Euclidean norms of the columns of the output matrix $A \cdot \text{diag}(D)$.

v If *jobv* = 'V', then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations.
 If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.
 If *jobv* = 'N', then *v* is not referenced.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

?gsvj1

Pre-processor for the routine ?gesvj, applies Jacobi rotations targeting only particular pivots.

Syntax

call `sgsvj1(jobv, m, n, nl, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)`

call `dgsvj1(jobv, m, n, nl, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)`

Include Files

- `mkl.fi`

Description

T

This routine is called from `?gesvj` as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as `?gesvj` does, but it targets only particular pivots and it does not check convergence (stopping criterion).

The routine `?gsvj1` applies few sweeps of Jacobi rotations in the column space of the input m -by- n matrix A . The pivot pairs are taken from the (1,2) off-diagonal block in the corresponding n -by- n Gram matrix A^*A . The block-entries (*tiles*) of the (1,2) off-diagonal block are marked by the `[x]`'s in the following scheme:

```
| *  *  *  [x] [x] [x] |
| *  *  *  [x] [x] [x] |
| *  *  *  [x] [x] [x] |
| [x] [x] [x] *  *  *  |
| [x] [x] [x] *  *  *  |
| [x] [x] [x] *  *  *  |
```

row-cycling in the nbl_r -by- $nbl_c[x]$ blocks, row-cyclic pivoting inside each `[x]` block

In terms of the columns of the matrix A , the first $n1$ columns are rotated 'against' the remaining $n-n1$ columns, trying to increase the angle between the corresponding subspaces. The off-diagonal block is $n1$ -by- $(n-n1)$ and it is tiled using quadratic tiles. The number of sweeps is specified by $nsweep$, and the orthogonality threshold is set by tol .

Input Parameters

<code>jobv</code>	CHARACTER*1. Must be 'V', 'A', or 'N'. Specifies whether the output from this routine is used to compute the matrix V . If <code>jobv = 'V'</code> , the product of the Jacobi rotations is accumulated by post-multiplying the n -by- n array v . If <code>jobv = 'A'</code> , the product of the Jacobi rotations is accumulated by post-multiplying the mv -by- n array v . If <code>jobv = 'N'</code> , the Jacobi rotations are not accumulated.
<code>m</code>	INTEGER. The number of rows of the input matrix A ($m \geq 0$).
<code>n</code>	INTEGER. The number of columns of the input matrix B ($m \geq n \geq 0$).
<code>n1</code>	INTEGER. Specifies the 2-by-2 block partition. The first $n1$ columns are rotated 'against' the remaining $n-n1$ columns of the matrix A .
<code>a</code>	REAL for <code>sgsvj1</code> DOUBLE PRECISION for <code>dgsvj1</code> . Arrays, DIMENSION ($lda, *$). Contains the m -by- n matrix A , such that $A \cdot \text{diag}(D)$ represents the input matrix. The second dimension of a must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
<code>d</code>	REAL for <code>sgsvj1</code> DOUBLE PRECISION for <code>dgsvj1</code> . Arrays, DIMENSION (n). Contains the diagonal matrix D that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry $A \cdot \text{diag}(D)$ represents the input matrix.
<code>sva</code>	REAL for <code>sgsvj1</code> DOUBLE PRECISION for <code>dgsvj1</code> .

	<p>Arrays, DIMENSION (n). Contains the Euclidean norms of the columns of the matrix $A*\text{diag}(D)$.</p>
<i>mv</i>	<p>INTEGER. The leading dimension of b; at least $\max(1, p)$.</p> <p>If $jobv = 'A'$, then mv rows of v are post-multiplied by a sequence of Jacobi rotations.</p> <p>If $jobv = 'N'$, then mv is not referenced .</p>
<i>v</i>	<p>REAL for <code>sgsvj1</code></p> <p>DOUBLE PRECISION for <code>dgsvj1</code>.</p> <p>Array, DIMENSION ($ldv, *$). The second dimension of a must be at least $\max(1, n)$.</p> <p>If $jobv = 'V'$, then n rows of v are post-multiplied by a sequence of Jacobi rotations.</p> <p>If $jobv = 'A'$, then mv rows of v are post-multiplied by a sequence of Jacobi rotations.</p> <p>If $jobv = 'N'$, then v is not referenced.</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array v; $ldv \geq 1$</p> <p>$ldv \geq n$ if $jobv = 'V'$;</p> <p>$ldv \geq mv$ if $jobv = 'A'$.</p>
<i>eps</i>	<p>REAL for <code>sgsvj1</code></p> <p>DOUBLE PRECISION for <code>dgsvj1</code>.</p> <p>The relative machine precision (epsilon) returned by the routine ?lamch.</p>
<i>sfmin</i>	<p>REAL for <code>sgsvj1</code></p> <p>DOUBLE PRECISION for <code>dgsvj1</code>.</p> <p>Value of safe minimum returned by the routine ?lamch.</p>
<i>tol</i>	<p>REAL for <code>sgsvj1</code></p> <p>DOUBLE PRECISION for <code>dgsvj1</code>.</p> <p>The threshold for Jacobi rotations. For a pair $A(:, p), A(:, q)$ of pivot columns, the Jacobi rotation is applied only if $\text{abs}(\cos(\text{angle}(A(:, p), A(:, q)))) > \text{tol}$.</p>
<i>nsweep</i>	<p>INTEGER.</p> <p>The number of sweeps of Jacobi rotations to be performed.</p>
<i>work</i>	<p>REAL for <code>sgsvj1</code></p> <p>DOUBLE PRECISION for <code>dgsvj1</code>.</p> <p>Workspace array, DIMENSION ($lwork$).</p>
<i>lwork</i>	<p>INTEGER. The size of the array $work$; at least $\max(1, m)$.</p>

Output Parameters

<i>a</i>	On exit, $A*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in <i>tol</i> and <i>nsweep</i> , respectively
<i>d</i>	On exit, $A*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in <i>tol</i> and <i>nsweep</i> , respectively.
<i>sva</i>	On exit, contains the Euclidean norms of the columns of the output matrix $A*\text{diag}(D)$.
<i>v</i>	<p>If <i>jobv</i> = 'V', then <i>n</i> rows of <i>v</i> are post-multiplied by a sequence of Jacobi rotations.</p> <p>If <i>jobv</i> = 'A', then <i>mv</i> rows of <i>v</i> are post-multiplied by a sequence of Jacobi rotations.</p> <p>If <i>jobv</i> = 'N', then <i>v</i> is not referenced.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

?sfrk

Performs a symmetric rank-*k* operation for matrix in RFP format.

Syntax

```
call ssfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call dsfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

Include Files

- mkl.fi

Description

The ?sfrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha A A^T + \beta C,$$

or

$$C := \alpha A^T A + \beta C,$$

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix in [rectangular full packed \(RFP\) format](#),

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

<i>transr</i>	<p>CHARACTER*1.</p> <p>if <i>transr</i> = 'N' or 'n', the normal form of RFP <i>C</i> is stored;</p> <p>if <i>transr</i> = 'T' or 't', the transpose form of RFP <i>C</i> is stored.</p>
<i>uplo</i>	<p>CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used.</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.</p>
<i>trans</i>	<p>CHARACTER*1. Specifies the operation:</p> <p>if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * A^T + \beta * C$;</p> <p>if <i>trans</i> = 'T' or 't', then $C := \alpha * A^T * A + \beta * C$;</p>
<i>n</i>	<p>INTEGER. Specifies the order of the matrix <i>C</i>. The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>A</i>, and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the matrix <i>A</i>.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <i>ssfrk</i></p> <p>DOUBLE PRECISION for <i>dsfrk</i></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <i>ssfrk</i></p> <p>DOUBLE PRECISION for <i>dsfrk</i></p> <p>Array, DIMENSION(<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>beta</i>	<p>REAL for <i>ssfrk</i></p> <p>DOUBLE PRECISION for <i>dsfrk</i></p> <p>Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>REAL for <i>ssfrk</i></p> <p>DOUBLE PRECISION for <i>dsfrk</i></p> <p>Array, size $(n * (n + 1) / 2)$. Before entry contains the symmetric matrix <i>C</i> in RFP format.</p>

Output Parameters

c If *trans* = 'N' or 'n', then *c* contains $C := \alpha * A * A' + \beta * C$;
 if *trans* = 'T' or 't', then *c* contains $C := \alpha * A' * A + \beta * C$;

?hfrk

Performs a Hermitian rank-*k* operation for matrix in RFP format.

Syntax

```
call chfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call zhfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

Include Files

- mkl.fi

Description

The ?hfrk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha * A * A^H + \beta * C,$$

or

$$C := \alpha * A^H * A + \beta * C,$$

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix in RFP format,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

transr CHARACTER*1.
 if *transr* = 'N' or 'n', the normal form of RFP *C* is stored;
 if *transr* = 'C' or 'c', the conjugate-transpose form of RFP *C* is stored.

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array *c* is used.
 If *uplo* = 'U' or 'u', then the upper triangular part of the array *c* is used.
 If *uplo* = 'L' or 'l', then the low triangular part of the array *c* is used.

trans CHARACTER*1. Specifies the operation:
 if *trans* = 'N' or 'n', then $C := \alpha * A * A^H + \beta * C$;
 if *trans* = 'C' or 'c', then $C := \alpha * A^H * A + \beta * C$.

n INTEGER. Specifies the order of the matrix *C*. The value of *n* must be at least zero.

<i>k</i>	<p>INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>a</i>, and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>a</i>.</p> <p>The value of <i>k</i> must be at least zero.</p>
<i>alpha</i>	<p>COMPLEX for chfrk DOUBLE COMPLEX for zhfrk</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>COMPLEX for chfrk DOUBLE COMPLEX for zhfrk</p> <p>Array, DIMENSION(<i>lda</i>, <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i>-by-<i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.</p>
<i>beta</i>	<p>COMPLEX for chfrk DOUBLE COMPLEX for zhfrk</p> <p>Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>COMPLEX for chfrk DOUBLE COMPLEX for zhfrk</p> <p>Array, size $(n*(n+1)/2)$. Before entry contains the Hermitian matrix <i>C</i> in RFP format.</p>

Output Parameters

<i>c</i>	<p>If <i>trans</i> = 'N' or 'n', then <i>c</i> contains $C := \alpha * A * A^H + \beta * C$; if <i>trans</i> = 'C' or 'c', then <i>c</i> contains $C := \alpha * A^H * A + \beta * C$;</p>
----------	--

?tfsm

Solves a matrix equation (one operand is a triangular matrix in RFP format).

Syntax

```
call stfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call dtfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ctfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ztfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
```

Include Files

- mkl.fi

Description

The `?tfsm` routines solve one of the following matrix equations:

$$\text{op}(A)*X = \text{alpha}*B,$$

or

$$X*\text{op}(A) = \text{alpha}*B,$$

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix in [rectangular full packed \(RFP\) format](#).

`op(A)` can be one of the following:

- `op(A) = A` or `op(A) = AT` for real flavors
- `op(A) = A` or `op(A) = AH` for complex flavors

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

<i>transr</i>	CHARACTER*1. if <i>transr</i> = 'N' or 'n', the normal form of RFP <i>A</i> is stored; if <i>transr</i> = 'T' or 't', the transpose form of RFP <i>A</i> is stored; if <i>transr</i> = 'C' or 'c', the conjugate-transpose form of RFP <i>A</i> is stored.
<i>side</i>	CHARACTER*1. Specifies whether <code>op(A)</code> appears on the left or right of <i>X</i> in the equation: if <i>side</i> = 'L' or 'l', then <code>op(A)*X = alpha*B</code> ; if <i>side</i> = 'R' or 'r', then <code>X*op(A) = alpha*B</code> .
<i>uplo</i>	CHARACTER*1. Specifies whether the RFP matrix <i>A</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the form of <code>op(A)</code> used in the matrix multiplication: if <i>trans</i> = 'N' or 'n', then <code>op(A) = A</code> ; if <i>trans</i> = 'T' or 't', then <code>op(A) = A'</code> ; if <i>trans</i> = 'C' or 'c', then <code>op(A) = conjg(A')</code> .
<i>diag</i>	CHARACTER*1. Specifies whether the RFP matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	INTEGER. Specifies the number of rows of <i>B</i> . The value of <i>m</i> must be at least zero.

<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for stfsm DOUBLE PRECISION for dtfsm COMPLEX for ctfsm DOUBLE COMPLEX for ztfsm Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	REAL for stfsm DOUBLE PRECISION for dtfsm COMPLEX for ctfsm DOUBLE COMPLEX for ztfsm Array, size $(n*(n+1)/2)$. Contains the matrix <i>A</i> in RFP format.
<i>b</i>	REAL for stfsm DOUBLE PRECISION for dtfsm COMPLEX for ctfsm DOUBLE COMPLEX for ztfsm Array, size $(1, ldb*n)$ Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the right-hand side matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

?lansf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.

Syntax

```
val = slansf(norm, transr, uplo, n, a, work)
```

```
val = dlansf(norm, transr, uplo, n, a, work)
```

Include Files

- mkl.fi

Description

T

The function `?lansf` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n real symmetric matrix A in the [rectangular full packed \(RFP\) format](#).

Input Parameters

<i>norm</i>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>transr</i>	<p>CHARACTER*1.</p> <p>Specifies whether the RFP format of matrix A is normal or transposed format.</p> <ul style="list-style-type: none"> If <i>transr</i> = 'N': RFP format is normal; if <i>transr</i> = 'T': RFP format is transposed.
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether the RFP matrix A came from upper or lower triangular matrix.</p> <ul style="list-style-type: none"> If <i>uplo</i> = 'U': RFP matrix A came from an upper triangular matrix; if <i>uplo</i> = 'L': RFP matrix A came from a lower triangular matrix.
<i>n</i>	<p>INTEGER. The order of the matrix A. $n \geq 0$.</p> <p>When $n = 0$, <code>?lansf</code> is set to zero.</p>
<i>a</i>	<p>REAL for <code>slansf</code></p> <p>DOUBLE PRECISION for <code>dlansf</code></p> <p>Array, DIMENSION $(n*(n+1)/2)$.</p> <p>The upper (if <i>uplo</i> = 'U') or lower (if <i>uplo</i> = 'L') part of the symmetric matrix A stored in RFP format.</p>
<i>work</i>	<p>REAL for <code>slansf</code>.</p> <p>DOUBLE PRECISION for <code>dlansf</code>.</p> <p>Workspace array, DIMENSION $(\max(1, lwork))$, where $lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.</p>

Output Parameters

<i>val</i>	<p>REAL for <code>slansf</code></p> <p>DOUBLE PRECISION for <code>dlansf</code></p>
------------	---

Value returned by the function.

?lanhf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.

Syntax

```
val = clanhf(norm, transr, uplo, n, a, work)
```

```
val = zlanhf(norm, transr, uplo, n, a, work)
```

Include Files

- mkl.fi

Description

The function ?lanhf returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n complex Hermitian matrix A in the [rectangular full packed \(RFP\) format](#).

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A . = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>transr</i>	CHARACTER*1. Specifies whether the RFP format of matrix A is normal or conjugate-transposed format. If <i>transr</i> = 'N': RFP format is normal; if <i>transr</i> = 'C': RFP format is conjugate-transposed.
<i>uplo</i>	CHARACTER*1. Specifies whether the RFP matrix A came from upper or lower triangular matrix. If <i>uplo</i> = 'U': RFP matrix A came from an upper triangular matrix; if <i>uplo</i> = 'L': RFP matrix A came from a lower triangular matrix.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lanhf is set to zero.

a COMPLEX for `clanhf`
 DOUBLE COMPLEX for `zlanhf`
 Array, DIMENSION ($n*(n+1)/2$).
 The upper (if `uplo = 'U'`) or lower (if `uplo = 'L'`) part of the Hermitian matrix *A* stored in RFP format.

work COMPLEX for `clanhf`.
 DOUBLE COMPLEX for `zlanhf`.
 Workspace array, DIMENSION ($\max(1, lwork)$), where
 $lwork \geq n$ when `norm = 'I'` or `'1'` or `'O'`; otherwise, *work* is not referenced.

Output Parameters

val COMPLEX for `clanhf`
 DOUBLE COMPLEX for `zlanhf`
 Value returned by the function.

?tfttp

Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).

Syntax

```
call stfttp( transr, uplo, n, arf, ap, info )
call dtfttp( transr, uplo, n, arf, ap, info )
call ctfttp( transr, uplo, n, arf, ap, info )
call ztfttp( transr, uplo, n, arf, ap, info )
```

Include Files

- `mkl.fi`

Description

The routine copies a triangular matrix *A* from the Rectangular Full Packed (RFP) format to the standard packed format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

transr CHARACTER*1.
 = 'N': *arf* is in the Normal format,
 = 'T': *arf* is in the Transpose format (for `stfttp` and `dtfttp`),
 = 'C': *arf* is in the Conjugate-transpose format (for `ctfttp` and `ztfttp`).

uplo CHARACTER*1.

Specifies whether A is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n INTEGER. The order of the matrix A . $n \geq 0$.

arf REAL for `stfttp`,
 DOUBLE PRECISION for `dtfttp`,
 COMPLEX for `ctfttp`,
 DOUBLE COMPLEX for `zftfttp`.

Array, size at least $\max(1, n*(n+1)/2)$.

On entry, the upper or lower triangular matrix A stored in the RFP format.

Output Parameters

ap REAL for `stfttp`,
 DOUBLE PRECISION for `dtfttp`,
 COMPLEX for `ctfttp`,
 DOUBLE COMPLEX for `zftfttp`.

Array, size at least $\max(1, n*(n+1)/2)$.

On exit, the upper or lower triangular matrix A , packed columnwise in a linear array.

The j -th column of A is stored in the array ap as follows:

if $uplo = 'U'$, $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,

if $uplo = 'L'$, $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

$info$ INTEGER. If $info = 0$, the execution is successful.
 If $info < 0$, the i -th parameter had an illegal value.
 If $info = -1011$, memory allocation error occurred.

?tfttr

Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR) .

Syntax

```
call stfttr( transr, uplo, n, arf, a, lda, info )
call dtfttr( transr, uplo, n, arf, a, lda, info )
call ctfttr( transr, uplo, n, arf, a, lda, info )
call zftfttr( transr, uplo, n, arf, a, lda, info )
```

Include Files

- `mkl.fi`

Description

The routine copies a triangular matrix A from the Rectangular Full Packed (RFP) format to the standard full format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>transr</i>	CHARACTER*1. = 'N': <i>arf</i> is in the Normal format, = 'T': <i>arf</i> is in the Transpose format (for <i>stfttr</i> and <i>dtfttr</i>), = 'C': <i>arf</i> is in the Conjugate-transpose format (for <i>ctfttr</i> and <i>zftttr</i>).
<i>uplo</i>	CHARACTER*1. Specifies whether A is upper or lower triangular: = 'U': A is upper triangular, = 'L': A is lower triangular.
<i>n</i>	INTEGER. The order of the matrices <i>arf</i> and a . $n \geq 0$.
<i>arf</i>	REAL for <i>stfttr</i> , DOUBLE PRECISION for <i>dtfttr</i> , COMPLEX for <i>ctfttr</i> , DOUBLE COMPLEX for <i>zftttr</i> . Array, size at least $\max(1, n*(n+1)/2)$. On entry, the upper or lower triangular matrix A stored in the RFP format.
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	REAL for <i>stfttr</i> , DOUBLE PRECISION for <i>dtfttr</i> , COMPLEX for <i>ctfttr</i> , DOUBLE COMPLEX for <i>zftttr</i> . Array, size (<i>lda</i> , *). On exit, the triangular matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the i -th parameter had an illegal value. If <i>info</i> = -1011, memory allocation error occurred.

?tpqrt2

Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.

Syntax

```
call stpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call dtpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ctpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ztpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call tpqrt2(a, b, t [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \left\{ \begin{array}{l} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{array} \right.$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-1)$ -by- n rectangular matrix $B1$ on top of an 1 -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \left\{ \begin{array}{l} \leftarrow (m-1) \times n \text{ rectangular} \\ \leftarrow 1 \times n \text{ upper trapezoidal} \end{array} \right.$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The matrix W contains the elementary reflectors $H(i)$ in the i th column below the diagonal (of A) in the $(n+m)$ -by- n input matrix C so that W can be represented as

$$W = \begin{bmatrix} I \\ V \end{bmatrix} \left\{ \begin{array}{l} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{array} \right.$$

Thus, V contains all of the information needed for W , and is returned in array b .

NOTE

V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \leftarrow \begin{matrix} (m-l) \times n \text{ rectangular} \\ l \times n \text{ upper trapezoidal} \end{matrix}$$

The columns of V represent the vectors which define the $H(i)$ s.

The $(m+n)$ -by- $(m+n)$ block reflector H is then given by

$H = I - W^* T^* W^T$ for real flavors, and

$H = I - W^* T^* W^H$ for complex flavors

where W^T is the transpose of W , W^H is the conjugate transpose of W , and T is the upper triangular factor of the block reflector.

Input Parameters

m	INTEGER. The total number of rows in the matrix B ($m \geq 0$).
n	INTEGER. The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
l	INTEGER. The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).
a, b	REAL for <code>stpqrt2</code> DOUBLE PRECISION for <code>dtpqrt2</code> COMPLEX for <code>ctpqrt2</code> COMPLEX*16 for <code>ztpqrt2</code> . Arrays: a , size (lda, n) contains the n -by- n upper triangular matrix A . b , size (ldb, n) , the pentagonal m -by- n matrix B . The first $(m-l)$ rows contain the rectangular $B1$ matrix, and the next l rows contain the upper trapezoidal $B2$ matrix.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least $\max(1, n)$.

Output Parameters

a	The elements on and above the diagonal of the array contain the upper triangular matrix R .
b	The pentagonal matrix V .
t	REAL for <code>stpqrt2</code> DOUBLE PRECISION for <code>dtpqrt2</code> COMPLEX for <code>ctpqrt2</code>

COMPLEX*16 for ztpqrt2.

Array, size (ldt, n).

The upper n -by- n upper triangular factor T of the block reflector.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info < 0$ and $info = -i$, the i th argument had an illegal value.

If $info = -1011$, memory allocation error occurred.

?tprfb

Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.

Syntax

```
call stprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

```
call dtprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

```
call ctprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

```
call ztprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

```
call tprfb(t, v, a, b[, direct][, storev][, side][, trans])
```

Include Files

- mkl.fi, lapack.f90

Description

The ?tprfb routine applies a real or complex "triangular-pentagonal" block reflector H , H^T , or H^H from either the left or the right to a real or complex matrix C , which is composed of two blocks A and B .

The block B is m -by- n . If $side = 'R'$, A is m -by- k , and if $side = 'L'$, A is of size k -by- n .

$$\begin{array}{ll}
 \text{direct} = 'F' & \text{direct} = 'B' \\
 \text{side} = 'R' & C = \begin{bmatrix} A & B \end{bmatrix} \quad C = \begin{bmatrix} B & A \end{bmatrix} \\
 \text{side} = 'L' & C = \begin{bmatrix} A \\ B \end{bmatrix} \quad C = \begin{bmatrix} B \\ A \end{bmatrix}
 \end{array}$$

The pentagonal matrix V is composed of a rectangular block $V1$ and a trapezoidal block $V2$. The size of the trapezoidal block is determined by the parameter l , where $0 \leq l \leq k$. if $l = k$, the $V2$ block of V is triangular; if $l = 0$, there is no trapezoidal block, thus $V = V1$ is rectangular.

<i>direct</i> ='F'	<i>direct</i> ='B'
<i>storev</i> ='C'	<i>storev</i> ='C'
$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$	$V = \begin{bmatrix} V2 \\ V1 \end{bmatrix}$
V2 is upper trapezoidal (first <i>l</i> rows of <i>k</i> -by- <i>k</i> upper triangular)	V2 is lower trapezoidal (last <i>l</i> rows of <i>k</i> -by- <i>k</i> lower triangular matrix)
<i>storev</i> ='R'	<i>storev</i> ='R'
$V = [V1 \quad V2]$	$V = [V2 \quad V1]$
V2 is lower trapezoidal (first <i>l</i> columns of <i>k</i> -by- <i>k</i> lower triangular matrix)	V2 is upper trapezoidal (last <i>l</i> columns of <i>k</i> -by- <i>k</i> upper triangular matrix)

<i>storev</i> ='C'	<i>side</i> ='L'	<i>side</i> ='R'
	V is <i>m</i> -by- <i>k</i>	V is <i>n</i> -by- <i>k</i>
	V2 is <i>l</i> -by- <i>k</i>	V2 is <i>l</i> -by- <i>k</i>
<i>storev</i> ='R'	V is <i>k</i> -by- <i>m</i>	V is <i>k</i> -by- <i>n</i>
	V2 is <i>k</i> -by- <i>l</i>	V2 is <i>k</i> -by- <i>l</i>

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply H , H^T , or H^H from the left, = 'R': apply H , H^T , or H^H from the right.
<i>trans</i>	CHARACTER*1. = 'N': apply H (no transpose), = 'T': apply H^T (transpose), = 'C': apply H^H (conjugate transpose).
<i>direct</i>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors: = 'F': $H = H(1) H(2) \dots H(k)$ (Forward), = 'B': $H = H(k) \dots H(2) H(1)$ (Backward).
<i>storev</i>	CHARACTER*1. Indicates how the vectors that define the elementary reflectors are stored: = 'C': Columns, = 'R': Rows.
<i>m</i>	INTEGER. The total number of rows in the matrix B ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in B ($n \geq 0$).

<i>k</i>	<p>INTEGER. The order of the matrix <i>T</i>, which is the number of elementary reflectors whose product defines the block reflector. ($k \geq 0$)</p>
<i>l</i>	<p>INTEGER. The order of the trapezoidal part of <i>V</i>. ($k \geq l \geq 0$).</p>
<i>v</i>	<p>REAL for <i>stprfb</i> DOUBLE PRECISION for <i>dtprfb</i> COMPLEX for <i>ctprfb</i> COMPLEX*16 for <i>ztprfb</i>. DIMENSION (<i>ldv</i>, <i>k</i>) if <i>storev</i> = 'C', DIMENSION (<i>ldv</i>, <i>m</i>) if <i>storev</i> = 'R' and <i>side</i> = 'L', DIMENSION (<i>ldv</i>, <i>n</i>) if <i>storev</i> = 'R' and <i>side</i> = 'R'. The pentagonal matrix <i>V</i>, which contains the elementary reflectors <i>H</i>(1), <i>H</i>(2), ..., <i>H</i>(<i>k</i>).</p>
<i>ldv</i>	<p>INTEGER. The leading dimension of the array <i>v</i>. If <i>storev</i> = 'C' and <i>side</i> = 'L', at least $\max(1, m)$. If <i>storev</i> = 'C' and <i>side</i> = 'R', at least $\max(1, n)$. If <i>storev</i> = 'R', at least <i>k</i>.</p>
<i>t</i>	<p>REAL for <i>stprfb</i> DOUBLE PRECISION for <i>dtprfb</i> COMPLEX for <i>ctprfb</i> COMPLEX*16 for <i>ztprfb</i>. Array size (<i>ldt</i>, <i>k</i>). The triangular <i>k</i>-by-<i>k</i> matrix <i>T</i> in the representation of the block reflector.</p>
<i>ldt</i>	<p>INTEGER. The leading dimension of the array <i>t</i> ($ldt \geq k$).</p>
<i>a</i>	<p>REAL for <i>stprfb</i> DOUBLE PRECISION for <i>dtprfb</i> COMPLEX for <i>ctprfb</i> COMPLEX*16 for <i>ztprfb</i>. DIMENSION (<i>lda</i>, <i>n</i>) if <i>side</i> = 'L', DIMENSION (<i>lda</i>, <i>k</i>) if <i>side</i> = 'R'. The <i>k</i>-by-<i>n</i> or <i>m</i>-by-<i>k</i> matrix <i>A</i>.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of the array <i>a</i>. If <i>side</i> = 'L', at least $\max(1, k)$. If <i>side</i> = 'R', at least $\max(1, m)$.</p>
<i>b</i>	<p>REAL for <i>stprfb</i> DOUBLE PRECISION for <i>dtprfb</i></p>

COMPLEX for `ctprfb`
 COMPLEX*16 for `ztpfrfb`.
 Array size (ldb, n), the m -by- n matrix B .

ldb INTEGER. The leading dimension of the array b ($ldb \geq \max(1, m)$).

work REAL for `stprfb`
 DOUBLE PRECISION for `dtprfb`
 COMPLEX for `ctprfb`
 COMPLEX*16 for `ztpfrfb`.
 DIMENSION ($ldwork, n$) if $side = 'L'$,
 DIMENSION ($ldwork, k$) if $side = 'R'$.
 Workspace array.

ldwork INTEGER. The leading dimension of the array *work*.
 If $side = 'L'$, at least k .
 If $side = 'R'$, at least m .

Output Parameters

a Contains the corresponding block of H^*C , H^T*C , H^H*C , C^*H , C^*H^T , or C^*H^H .

b Contains the corresponding block of H^*C , H^T*C , H^H*C , C^*H , C^*H^T , or C^*H^H .

info INTEGER. If $info = 0$, the execution is successful.
 If $info < 0$, the i -th parameter had an illegal value.
 If $info = -1011$, memory allocation error occurred.

?tpdff

Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).

Syntax

```
call stpdff( transr, uplo, n, ap, arf, info )
call dtpdff( transr, uplo, n, ap, arf, info )
call ctpdff( transr, uplo, n, ap, arf, info )
call ztpdff( transr, uplo, n, ap, arf, info )
```

Include Files

- `mkl.fi`

Description

The routine copies a triangular matrix A from the standard packed format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>transr</i>	<p>CHARACTER*1.</p> <p>= 'N': <i>arf</i> must be in the Normal format,</p> <p>= 'T': <i>arf</i> must be in the Transpose format (for <i>stpttf</i> and <i>dtpttf</i>),</p> <p>= 'C': <i>arf</i> must be in the Conjugate-transpose format (for <i>ctpttf</i> and <i>ztpttf</i>).</p>
<i>uplo</i>	<p>CHARACTER*1.</p> <p>Specifies whether <i>A</i> is upper or lower triangular:</p> <p>= 'U': <i>A</i> is upper triangular,</p> <p>= 'L': <i>A</i> is lower triangular.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>A</i>. $n \geq 0$.</p>
<i>ap</i>	<p>REAL for <i>stpttf</i>,</p> <p>DOUBLE PRECISION for <i>dtpttf</i>,</p> <p>COMPLEX for <i>ctpttf</i>,</p> <p>DOUBLE COMPLEX for <i>ztpttf</i>.</p> <p>Array, size at least $\max(1, n*(n+1)/2)$.</p> <p>On entry, the upper or lower triangular matrix <i>A</i>, packed columnwise in a linear array.</p> <p>The <i>j</i>-th column of <i>A</i> is stored in the array <i>ap</i> as follows:</p> <p>if <i>uplo</i> = 'U', $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,</p> <p>if <i>uplo</i> = 'L', $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.</p>

Output Parameters

<i>arf</i>	<p>REAL for <i>stpttf</i>,</p> <p>DOUBLE PRECISION for <i>dtpttf</i>,</p> <p>COMPLEX for <i>ctpttf</i>,</p> <p>DOUBLE COMPLEX for <i>ztpttf</i>.</p> <p>Array, size at least $\max(1, n*(n+1)/2)$.</p> <p>On exit, the upper or lower triangular matrix <i>A</i> stored in the RFP format.</p>
<i>info</i>	<p>INTEGER.</p> <p>=0: successful exit,</p> <p>< 0: if <i>info</i> = <i>-i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = -1011, memory allocation error occurred.</p>

?tptr

Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).

Syntax

```
call stpttr( uplo, n, ap, a, lda, info )
call dtpttr( uplo, n, ap, a, lda, info )
call ctpttr( uplo, n, ap, a, lda, info )
call ztpttr( uplo, n, ap, a, lda, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix A from the standard packed format to the standard full format.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether A is upper or lower triangular: = 'U': A is upper triangular, = 'L': A is lower triangular.
<i>n</i>	INTEGER. The order of the matrices ap and a . $n \geq 0$.
<i>ap</i>	REAL for stpttr, DOUBLE PRECISION for dtpttr, COMPLEX for ctpttr, DOUBLE COMPLEX for ztpttr. On entry, the upper or lower triangular matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows: if $uplo = 'U'$, $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$, if $uplo = 'L'$, $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.
<i>lda</i>	INTEGER. The leading dimension of the array a . $lda \geq \max(1,n)$.

Output Parameters

<i>a</i>	REAL for stpttr, DOUBLE PRECISION for dtpttr, COMPLEX for ctpttr, DOUBLE COMPLEX for ztpttr. Array, size (lda , *).
----------	--

On exit, the triangular matrix A . If $uplo = 'U'$, the leading n -by- n upper triangular part of the array a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of the array a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.
 If $info = -1011$, memory allocation error occurred.

?trttf

Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).

Syntax

```
call strttf( transr, uplo, n, a, lda, arf, info )
call dtrttf( transr, uplo, n, a, lda, arf, info )
call ctrttf( transr, uplo, n, a, lda, arf, info )
call ztrttf( transr, uplo, n, a, lda, arf, info )
```

Include Files

- mkl.fi

Description

The routine copies a triangular matrix A from the standard full format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

transr CHARACTER*1.
 = 'N': arf must be in the Normal format,
 = 'T': arf must be in the Transpose format (for `strttf` and `dtrttf`),
 = 'C': arf must be in the Conjugate-transpose format (for `ctrttf` and `ztrttf`).

uplo CHARACTER*1.
 Specifies whether A is upper or lower triangular:
 = 'U': A is upper triangular,
 = 'L': A is lower triangular.

n INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for `strttf`,

DOUBLE PRECISION for `dtrttf`,
 COMPLEX for `ctrttf`,
 DOUBLE COMPLEX for `ztrttf`.

Array, size (lda, n) .

On entry, the triangular matrix A . If `uplo = 'U'`, the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced. If `uplo = 'L'`, the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.

`lda` INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

`arf` REAL for `strttf`,
 DOUBLE PRECISION for `dtrttf`,
 COMPLEX for `ctrttf`,
 DOUBLE COMPLEX for `ztrttf`.

Array, size at least $\max(1, n*(n+1)/2)$.

On exit, the upper or lower triangular matrix A stored in the RFP format.

`info` INTEGER. If `info = 0`, the execution is successful.
 If `info < 0`, the i -th parameter had an illegal value.
 If `info = -1011`, memory allocation error occurred.

?trttp

Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).

Syntax

```
call strttp( uplo, n, a, lda, ap, info )
call dtrttp( uplo, n, a, lda, ap, info )
call ctrttp( uplo, n, a, lda, ap, info )
call ztrttp( uplo, n, a, lda, ap, info )
```

Include Files

- `mkl.fi`

Description

The routine copies a triangular matrix A from the standard full format to the standard packed format.

Input Parameters

`uplo` CHARACTER*1.

Specifies whether A is upper or lower triangular:

= 'U': A is upper triangular,

= 'L': A is lower triangular.

n INTEGER. The order of the matrix A , $n \geq 0$.

a REAL for `strttp`,
 DOUBLE PRECISION for `dtrttp`,
 COMPLEX for `ctrttp`,
 DOUBLE COMPLEX for `ztrttp`.

Array, size (lda, n) .

On entry, the triangular matrix A . If `uplo` = 'U', the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced. If `uplo` = 'L', the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

ap REAL for `strttp`,
 DOUBLE PRECISION for `dtrttp`,
 COMPLEX for `ctrttp`,
 DOUBLE COMPLEX for `ztrttp`.

Array, size at least $\max(1, n*(n+1)/2)$.

On exit, the upper or lower triangular matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows:

if `uplo` = 'U', $ap(i + (j-1)*j/2) = A(i, j)$ for $1 \leq i \leq j$,

if `uplo` = 'L', $ap(i + (j-1)*(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.

$info$ INTEGER. If $info = 0$, the execution is successful.
 If $info < 0$, the i -th parameter had an illegal value.
 If $info = -1011$, memory allocation error occurred.

?pstf2

Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.

Syntax

```
call spstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
```

call `zpstf2(uplo, n, a, lda, piv, rank, tol, work, info)`

Include Files

- `mkl.fi`

Description

The real flavors `spstf2` and `dpstf2` compute the Cholesky factorization with complete pivoting of a real symmetric positive semi-definite matrix A . The complex flavors `cpstf2` and `zpstf2` compute the Cholesky factorization with complete pivoting of a complex Hermitian positive semi-definite matrix A . The factorization has the form:

$$P^T * A * P = U^T * U, \text{ if } uplo = 'U' \text{ for real flavors,}$$

$$P^T * A * P = U^H * U, \text{ if } uplo = 'U' \text{ for complex flavors,}$$

$$P^T * A * P = L * L^T, \text{ if } uplo = 'L' \text{ for real flavors,}$$

$$P^T * A * P = L * L^H, \text{ if } uplo = 'L' \text{ for complex flavors,}$$

where U is an upper triangular matrix and L is lower triangular, and P is stored as vector `piv`.

This algorithm does not check that A is positive semi-definite. This version of the algorithm calls [level 2 BLAS](#).

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix A is stored: = 'U': Upper triangular, = 'L': Lower triangular.
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$.
<code>a</code>	REAL for <code>spstf2</code> , DOUBLE PRECISION for <code>dpstf2</code> , COMPLEX for <code>cpstf2</code> , DOUBLE COMPLEX for <code>zpstf2</code> . Array, DIMENSION (<code>lda</code> , *). On entry, the symmetric matrix A . If <code>uplo</code> = 'U', the leading n -by- n upper triangular part of the array <code>a</code> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <code>a</code> is not referenced. If <code>uplo</code> = 'L', the leading n -by- n lower triangular part of the array <code>a</code> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <code>a</code> is not referenced.
<code>tol</code>	REAL for <code>spstf2</code> and <code>cpstf2</code> , DOUBLE PRECISION for <code>dpstf2</code> and <code>zpstf2</code> . A user-defined tolerance. If <code>tol</code> < 0, $n * ulp * \max(A(k,k))$ will be used (<code>ulp</code> is the Unit in the Last Place, or Unit of Least Precision). The algorithm terminates at the $(k - 1)$ -st step if the pivot is not greater than <code>tol</code> .

lda INTEGER. The leading dimension of the matrix *A*. $lda \geq \max(1,n)$.

work REAL for *spstf2* and *cpstf2*,
 DOUBLE PRECISION for *dpstf2* and *zpstf2*.
 Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

piv INTEGER. Array. DIMENSION at least $\max(1,n)$.
piv is such that the non-zero entries are $P(piv(k), k) = 1$.

a On exit, if *info* = 0, the factor U or L from the Cholesky factorization stored the same way as the matrix *A* is stored on entry.

rank INTEGER.
 The rank of *A*, determined by the number of steps the algorithm completed.

info INTEGER.
 < 0: if *info* = -*k*, the *k*-th parameter had an illegal value,
 =0: the algorithm completed successfully,
 > 0: the matrix *A* is rank-deficient with the computed rank, returned in *rank*, or indefinite.

dlat2s

Converts a double-precision triangular matrix to a single-precision triangular matrix.

Syntax

```
call dlat2s( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

This routine converts a double-precision triangular matrix *A* to a single-precision triangular matrix *SA*. *dlat2s* checks that all the elements of *A* are between *-RMAX* and *RMAX*, where *RMAX* is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

uplo CHARACTER*1.
 Specifies whether the matrix *A* is upper or lower triangular:
 = 'U': *A* is upper triangular,
 = 'L': *A* is lower triangular.

n INTEGER. The number of rows and columns of the matrix *A*. $n \geq 0$.

<i>a</i>	DOUBLE PRECISION. Array, DIMENSION (<i>lda</i> , *). On entry, the <i>n</i> -by- <i>n</i> triangular matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,n)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> . $ldsa \geq \max(1,n)$.

Output Parameters

<i>sa</i>	REAL. Array, DIMENSION (<i>ldsa</i> , *). Only the part of <i>sa</i> determined by <i>uplo</i> is referenced. On exit, <ul style="list-style-type: none"> • if <i>info</i> = 0, the <i>n</i>-by-<i>n</i> triangular matrix <i>SA</i>, • if <i>info</i> > 0, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified.
<i>info</i>	INTEGER. =0: successful exit, > 0: an element of the matrix <i>A</i> is greater than the single-precision overflow threshold; in this case, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified on exit.

zlat2c

Converts a double complex triangular matrix to a complex triangular matrix.

Syntax

```
call zlat2c( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- mkl.fi

Description

This routine is declared in `mkl_lapack.fi`.

The routine converts a DOUBLE COMPLEX triangular matrix *A* to a COMPLEX triangular matrix *SA*. `zlat2c` checks that the real and complex parts of all the elements of *A* are between *-RMAX* and *RMAX*, where *RMAX* is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
-------------	---

<i>n</i>	INTEGER. The number of rows and columns in the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	DOUBLE COMPLEX. Array, DIMENSION (<i>lda</i> , *). On entry, the <i>n</i> -by- <i>n</i> triangular matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,n)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> . $ldsa \geq \max(1,n)$.

Output Parameters

<i>sa</i>	COMPLEX. Array, DIMENSION (<i>ldsa</i> , *). Only the part of <i>sa</i> determined by <i>uplo</i> is referenced. On exit, <ul style="list-style-type: none"> if <i>info</i> = 0, the <i>n</i>-by-<i>n</i> triangular matrix <i>sa</i>, if <i>info</i> > 0, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified.
<i>info</i>	INTEGER. =0: successful exit, > 0: the real or complex part of an element of the matrix <i>A</i> is greater than the single-precision overflow threshold; in this case, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified on exit.

?lacp2

Copies all or part of a real two-dimensional array to a complex array.

Syntax

```
call clacp2( uplo, m, n, a, lda, b, ldb )
```

```
call zlacp2( uplo, m, n, a, lda, b, ldb )
```

Include Files

- mkl.fi

Description

The routine copies all or part of a real matrix *A* to another matrix *B*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix <i>A</i> to be copied to <i>B</i> . If <i>uplo</i> = 'U', the upper triangular part of <i>A</i> ; if <i>uplo</i> = 'L', the lower triangular part of <i>A</i> .
-------------	--

	Otherwise, all of the matrix A is copied.
m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
a	REAL for <code>clacp2</code> DOUBLE PRECISION for <code>zlacp2</code> Array, size at least (lda, n) , contains the m -by- n matrix A . If $uplo = 'U'$, only the upper triangle or trapezoid is accessed; if $uplo = 'L'$, only the lower triangle or trapezoid is accessed.
lda	INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.
ldb	INTEGER. The leading dimension of the output array b ; $ldb \geq \max(1, m)$.

Output Parameters

b	COMPLEX for <code>clacp2</code> DOUBLE COMPLEX for <code>zlacp2</code> . Array, size (ldb, n) . On exit, $B = A$ in the locations specified by $uplo$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info < 0$, the i -th parameter had an illegal value. If $info = -1011$, memory allocation error occurred.

?la_gbamv

Performs a matrix-vector operation to calculate error bounds.

Syntax

```
call sla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call dla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call cla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call zla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
```

Include Files

- `mkl.fi`

Description

The `?la_gbamv` function performs one of the matrix-vector operations defined as

$$y := \alpha \cdot \text{abs}(A) \cdot \text{abs}(x) + \beta \cdot \text{abs}(y),$$

or

$$y := \alpha \cdot \text{abs}(A)^T \cdot \text{abs}(x) + \beta \cdot \text{abs}(y),$$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n matrix, with kl sub-diagonals and ku super-diagonals.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

<i>trans</i>	<p>INTEGER. Specifies the operation to be performed:</p> <p>If <i>trans</i> = 'BLAS_NO_TRANS', then $y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y)$</p> <p>If <i>trans</i> = 'BLAS_TRANS', then $y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y)$</p> <p>If <i>trans</i> = 'BLAS_CONJ_TRANS', then $y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y)$</p> <p>The parameter is unchanged on exit.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix A.</p> <p>The value of m must be at least zero. Unchanged on exit.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix A.</p> <p>The value of n must be at least zero. Unchanged on exit.</p>
<i>kl</i>	<p>INTEGER. Specifies the number of sub-diagonals within the band of A.</p> <p>$kl \geq 0$.</p>
<i>ku</i>	<p>INTEGER. Specifies the number of super-diagonals within the band of A.</p> <p>$ku \geq 0$.</p>
<i>alpha</i>	<p>REAL for <code>sla_gbamv</code> and <code>cla_gbamv</code></p> <p>DOUBLE PRECISION for <code>dla_gbamv</code> and <code>zla_gbamv</code></p> <p>Specifies the scalar <i>alpha</i>. Unchanges on exit.</p>
<i>ab</i>	<p>REAL for <code>sla_gbamv</code></p> <p>DOUBLE PRECISION for <code>dla_gbamv</code></p> <p>COMPLEX for <code>cla_gbamv</code></p> <p>DOUBLE COMPLEX for <code>zla_gbamv</code></p> <p>Array, DIMENSION(<i>ldab</i>, *).</p> <p>Before entry, the leading m-by-n part of the array <i>ab</i> must contain the matrix of coefficients. The second dimension of <i>ab</i> must be at least $\max(1, n)$. Unchanged on exit.</p>

<i>ldab</i>	INTEGER. Specifies the leading dimension of <i>ab</i> as declared in the calling (sub)program. The value of <i>ldab</i> must be at least $\max(1, m)$. Unchanged on exit.
<i>x</i>	REAL for <i>sla_gbamv</i> DOUBLE PRECISION for <i>dla_gbamv</i> COMPLEX for <i>cla_gbamv</i> DOUBLE COMPLEX for <i>zla_gbamv</i> Array, DIMENSION $(1 + (n - 1) * \text{abs}(incx))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (m - 1) * \text{abs}(incx))$ otherwise. Before entry, the incremented array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>beta</i>	REAL for <i>sla_gbamv</i> and <i>cla_gbamv</i> DOUBLE PRECISION for <i>dla_gbamv</i> and <i>zla_gbamv</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is zero, you do not need to set <i>y</i> on input.
<i>y</i>	REAL for <i>sla_gbamv</i> and <i>cla_gbamv</i> DOUBLE PRECISION for <i>dla_gbamv</i> and <i>zla_gbamv</i> Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(incy))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry with <i>beta</i> non-zero, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero. Unchanged on exit.

Output Parameters

<i>y</i>	Updated vector <i>y</i> .
----------	---------------------------

?la_gbrcond

Estimates the Skeel condition number for a general banded matrix.

Syntax

call *sla_gbrcond*(*trans*, *n*, *kl*, *ku*, *ab*, *ldab*, *afb*, *ldafb*, *ipiv*, *cmode*, *c*, *info*, *work*, *iwork*)

call `dla_gbrcond(trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, cmode, c, info, work, iwork)`

Include Files

- `mkl.fi`

Description

The function estimates the Skeel condition number of

$op(A) * op2(C)$

where

the `cmode` parameter determines `op2` as follows:

<code>cmode</code> Value	<code>op2(C)</code>
1	C
0	I
-1	$inv(C)$

The Skeel condition number

$cond(A) = norminf(|inv(A)| |A|)$

is computed by computing scaling factors R such that

$diag(R) * A * op2(C)$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<code>trans</code>	<p>CHARACTER*1. Must be 'N' or 'T' or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <code>trans</code> = 'N', the system has the form $A * X = B$.</p> <p>If <code>trans</code> = 'T', the system has the form $A^T * X = B$.</p> <p>If <code>trans</code> = 'C', the system has the form $A^H * X = B$.</p>
<code>n</code>	<p>INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.</p>
<code>kl</code>	<p>INTEGER. The number of subdiagonals within the band of A; $kl \geq 0$.</p>
<code>ku</code>	<p>INTEGER. The number of superdiagonals within the band of A; $ku \geq 0$.</p>
<code>ab, afb, c, work</code>	<p>REAL for <code>sla_gbrcond</code></p> <p>DOUBLE PRECISION for <code>dla_gbrcond</code></p> <p>Arrays:</p> <p><code>ab(ldab,*)</code> contains the original band matrix A stored in rows from 1 to $kl + ku + 1$. The j-th column of A is stored in the j-th column of the array <code>ab</code> as follows:</p> <p>$ab(ku+1+i-j, j) = A(i, j)$</p> <p>for</p>

$\max(1, j-ku) \leq i \leq \min(n, j+kl)$

afb(ldafb,)* contains details of the LU factorization of the band matrix *A*, as returned by *?gbtrf*. *U* is stored as an upper triangular band matrix with *kl+ku* superdiagonals in rows 1 to *kl+ku+1*, and the multipliers used during the factorization are stored in rows *kl+ku+2* to *2*kl+ku+1*.

c, DIMENSION *n*. The vector *C* in the formula $\text{op}(A) * \text{op2}(C)$.

work is a workspace array of DIMENSION $(5*n)$.

The second dimension of *ab* and *afb* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*. $ldab \geq kl+ku+1$.

ldaafb INTEGER. The leading dimension of *afb*. $ldaafb \geq 2*kl+ku+1$.

ipiv INTEGER.

Array with DIMENSION *n*. The pivot indices from the factorization $A = P*L*U$ as computed by *?gbtrf*. Row *i* of the matrix was interchanged with row *ipiv(i)*.

cmode INTEGER. Determines $\text{op2}(C)$ in the formula $\text{op}(A) * \text{op2}(C)$ as follows:

If *cmode* = 1, $\text{op2}(C) = C$.

If *cmode* = 0, $\text{op2}(C) = I$.

If *cmode* = -1, $\text{op2}(C) = \text{inv}(C)$.

iwork INTEGER. Workspace array with DIMENSION *n*.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

[?gbtrf](#)

[?la_gbrcond_c](#)

Computes the infinity norm condition number of $\text{op}(A)\text{inv}(\text{diag}(C))$ for general banded matrices.*

Syntax

```
call cla_gbrcond_c( trans, n, kl, ku, ab, ldab, afb, ldaafb, ipiv, c, capply, info,
work, rwork )
```

```
call zla_gbrcond_c( trans, n, kl, ku, ab, ldab, afb, ldaafb, ipiv, c, capply, info,
work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{inv}(\text{diag}(c))$

where the c is a REAL vector for `cla_gbrcond_c` and a DOUBLE PRECISION vector for `zla_gbrcond_c`.

Input Parameters

<i>trans</i>	<p>CHARACTER*1. Must be 'N' or 'T' or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A*X = B$ (No transpose)</p> <p>If <i>trans</i> = 'T', the system has the form $A^T*X = B$ (Transpose)</p> <p>If <i>trans</i> = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)</p>
<i>n</i>	<p>INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.</p>
<i>kl</i>	<p>INTEGER. The number of subdiagonals within the band of A; $kl \geq 0$.</p>
<i>ku</i>	<p>INTEGER. The number of superdiagonals within the band of A; $ku \geq 0$.</p>
<i>ab,afb,work</i>	<p>COMPLEX for <code>cla_gbrcond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_gbrcond_c</code></p> <p>Arrays:</p> <p><i>ab</i>(<i>ldab</i>,*) contains the original band matrix A stored in rows from 1 to $kl + ku + 1$. The j-th column of A is stored in the j-th column of the array <i>ab</i> as follows:</p> $ab(ku+1+i-j, j) = A(i, j)$ <p>for</p> $\max(1, j-ku) \leq i \leq \min(n, j+kl)$ <p><i>afb</i>(<i>ldafb</i>,*) contains details of the LU factorization of the band matrix A, as returned by <code>?gbtrf</code>. U is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$.</p> <p><i>work</i> is a workspace array of DIMENSION (5*n).</p> <p>The second dimension of <i>ab</i> and <i>afb</i> must be at least $\max(1, n)$.</p>
<i>ldab</i>	<p>INTEGER. The leading dimension of the array <i>ab</i>. $ldab \geq kl+ku+1$.</p>
<i>ldafb</i>	<p>INTEGER. The leading dimension of <i>afb</i>. $ldafb \geq 2*kl+ku+1$.</p>
<i>ipiv</i>	<p>INTEGER.</p> <p>Array with DIMENSION n. The pivot indices from the factorization $A = P*L*U$ as computed by <code>?gbtrf</code>. Row i of the matrix was interchanged with row <i>ipiv</i>(i).</p>
<i>c, rwork</i>	<p>REAL for <code>cla_gbrcond_c</code></p> <p>DOUBLE PRECISION for <code>zla_gbrcond_c</code></p> <p>Array <i>c</i> with DIMENSION n. The vector c in the formula</p> $\text{op}(A) * \text{inv}(\text{diag}(c)).$

Array *rwork* with DIMENSION n is a workspace.

caply LOGICAL. If `.TRUE.`, then the function uses the vector *c* from the formula $\text{op}(A) * \text{inv}(\text{diag}(c))$.

Output Parameters

info INTEGER.
If *info* = 0, the execution is successful.
If *i* > 0, the *i*-th parameter is invalid.

See Also

?gbtrf

?la_gbrcond_x

Computes the infinity norm condition number of $\text{op}(A)\text{diag}(x)$ for general banded matrices.*

Syntax

```
call cla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work,
rwork )
```

```
call zla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work,
rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the *x* is a COMPLEX vector for `cla_gbrcond_x` and a DOUBLE COMPLEX vector for `zla_gbrcond_x`.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form $A*X = B$ (No transpose)
If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose)
If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

kl INTEGER. The number of subdiagonals within the band of *A*; $kl \geq 0$.

ku INTEGER. The number of superdiagonals within the band of *A*; $ku \geq 0$.

ab, *afb*, *x*, *work* COMPLEX for `cla_gbrcond_x`
 DOUBLE COMPLEX for `zla_gbrcond_x`

Arrays:

ab(*ldab*,*) contains the original band matrix *A* stored in rows from 1 to *kl* + *ku* + 1. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

$$ab(ku+1+i-j, j) = A(i, j)$$

for

$$\max(1, j-ku) \leq i \leq \min(n, j+kl)$$

afb(*ldafb*,*) contains details of the LU factorization of the band matrix *A*, as returned by `?gbtrf`. *U* is stored as an upper triangular band matrix with *kl*+*ku* superdiagonals in rows 1 to *kl*+*ku*+1, and the multipliers used during the factorization are stored in rows *kl*+*ku*+2 to *2*kl*+*ku*+1.

x, DIMENSION *n*. The vector *x* in the formula $op(A) * diag(x)$.

work is a workspace array of DIMENSION (2**n*).

The second dimension of *ab* and *afb* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*. *ldab* ≥ *kl*+*ku*+1.

ldafb INTEGER. The leading dimension of *afb*. *ldafb* ≥ 2**kl*+*ku*+1.

ipiv INTEGER.

Array with DIMENSION *n*. The pivot indices from the factorization $A = P*L*U$ as computed by `?gbtrf`. Row *i* of the matrix was interchanged with row *ipiv*(*i*).

rwork REAL for `cla_gbrcond_x`
 DOUBLE PRECISION for `zla_gbrcond_x`

Array *rwork* with DIMENSION *n* is a workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

[?gbtrf](#)

[?la_gbrfsx_extended](#)

Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call sla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )
```

```
call dla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )
```

```
call cla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )
```

```
call zla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )
```

Include Files

- mkl.fi

Description

The `?la_gbrfsx_extended` subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?gbrfsx` routine calls `?la_gbrfsx_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use `?la_gbrfsx_extended` to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

<i>prec_type</i>	INTEGER. Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code> , where <code>p</code> is a CHARACTER and: If <code>p = 'S'</code> : Single. If <code>p = 'D'</code> : Double. If <code>p = 'I'</code> : Indigenous. If <code>p = 'X', 'E'</code> : Extra.
<i>trans_type</i>	INTEGER. Specifies the transposition operation on <i>A</i> . The value is defined by <code>ilatrans(t)</code> , where <code>t</code> is a CHARACTER and: If <code>t = 'N'</code> : No transpose. If <code>t = 'T'</code> : Transpose. If <code>t = 'C'</code> : Conjugate Transpose.

<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrix <i>B</i> .
<i>ab, afb, b, y</i>	<p>REAL for <code>sla_gbrfsx_extended</code> DOUBLE PRECISION for <code>dla_gbrfsx_extended</code> COMPLEX for <code>cla_gbrfsx_extended</code> DOUBLE COMPLEX for <code>zla_gbrfsx_extended</code>.</p> <p>Arrays: <code>ab(ldab,*)</code>, <code>afb(ldafb,*)</code>, <code>b(l db,*)</code>, <code>y(ldy,*)</code>.</p> <p>The array <i>ab</i> contains the original <i>n</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>ab</i> must be at least $\max(1, n)$.</p> <p>The array <i>afb</i> contains the factors <i>L</i> and <i>U</i> from the factorization $A = P * L * U$ as computed by <code>?gbtrf</code>. The second dimension of <i>afb</i> must be at least $\max(1, n)$.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p> <p>The array <i>y</i> on entry contains the solution matrix <i>X</i> as computed by <code>?gbtrs</code>. The second dimension of <i>y</i> must be at least $\max(1, nrhs)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; $ldab \geq \max(1, n)$.
<i>ldafb</i>	INTEGER. The leading dimension of the array <i>afb</i> ; $ldafb \geq \max(1, n)$.
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices from the factorization $A = P * L * U$ as computed by <code>?gbtrf</code>; row <i>i</i> of the matrix was interchanged with row <code>ipiv(i)</code>.</p>
<i>colequ</i>	LOGICAL. If <code>colequ = .TRUE.</code> , column equilibration was done to <i>A</i> before calling this routine. This is needed to compute the solution and error bounds correctly.
<i>c</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p><i>c</i> contains the column scale factors for <i>A</i>. If <code>colequ = .FALSE.</code>, <i>c</i> is not accessed.</p> <p>If <i>c</i> is input, each element of <i>c</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>

ldb INTEGER. The leading dimension of the array *b*; $ldb \geq \max(1, n)$.

ldy INTEGER. The leading dimension of the array *y*; $ldy \geq \max(1, n)$.

n_norms INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.
 If $n_norms \geq 1$, returns normwise error bounds.
 If $n_norms \geq 2$, returns componentwise error bounds.

err_bnds_norm REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm*(*i*, :) corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm*(:, *err*) contains the following three fields:

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition

numbers are $1 / (\text{norm}(1 / z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

err_bnds_comp

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params*(3) = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first (*:, n_err_bnds*) entries are returned.

The first index in *err_bnds_comp*($i, :$) corresponds to the i -th right-hand side.

The second index in *err_bnds_comp*($:, err$) contains the following three fields:

- err*=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors.
- err*=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
- err*=3 Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold

$\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

res, dy, y_tail

REAL for `sla_gbrfsx_extended`
 DOUBLE PRECISION for `dla_gbrfsx_extended`
 COMPLEX for `cla_gbrfsx_extended`
 DOUBLE COMPLEX for `zla_gbrfsx_extended`.

Workspace arrays of DIMENSION n .

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION n .

rcond

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(\text{dx}_{\{i+1\}}) < r\text{thresh} * \text{norm}(\text{dx}_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$0 < r\text{thresh} \leq 1$.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to start considering componentwise convergence.

Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

REAL for `sla_gbrfsx_extended`

DOUBLE PRECISION for `dla_gbrfsx_extended`

COMPLEX for `cla_gbrfsx_extended`

DOUBLE COMPLEX for `zla_gbrfsx_extended`.

The improved solution matrix Y .

berr_out

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for right-hand-side j from the formula

$\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B)) (i))$

where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z . This is computed by `?la_lin_berr`.

err_bnds_norm,
err_bnds_comp

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array $(1:nrhs, 2)$. The other elements are kept unchanged.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = $-i$, the i -th parameter had an illegal value.

See Also

[?gbrfsx](#)

[?gbtrf](#)

[?gbtrs](#)

?lamch
 ilaprec
 ilatrans
 ?la_lin_berr

?la_gbrpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a general band matrix.

Syntax

```
call sla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call dla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call cla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call zla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
```

Include Files

- mkl.fi

Description

The ?la_gbrpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>n</i>	INTEGER. The number of linear equations, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ncols</i>	INTEGER. The number of columns of the matrix <i>A</i> ; $ncols \geq 0$.
<i>ab, afb</i>	REAL for sla_gbrpvgrw DOUBLE PRECISION for dla_gbrpvgrw COMPLEX for cla_gbrpvgrw DOUBLE COMPLEX for zla_gbrpvgrw. Arrays: <i>ab(ldab,*)</i> , <i>afb(ldafb,*)</i> . <i>ab</i> contains the original band matrix <i>A</i> (see Matrix Storage Schemes) stored in rows from 1 to $kl + ku + 1$. The <i>j</i> -th column of <i>A</i> is stored in the <i>j</i> -th column of the array <i>ab</i> as follows: $ab(ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$

afb contains details of the LU factorization of the band matrix *A*, as returned by `?gbtrf`. *U* is stored as an upper triangular band matrix with k_l+k_u superdiagonals in rows 1 to k_l+k_u+1 , and the multipliers used during the factorization are stored in rows k_l+k_u+2 to $2*k_l+k_u+1$.

ldab INTEGER. The leading dimension of *ab*; $ldab \geq k_l+k_u+1$.

ldafb INTEGER. The leading dimension of *afb*; $ldafb \geq 2*k_l+k_u+1$.

See Also

[?gbtrf](#)

?la_geamv

Computes a matrix-vector product using a general matrix to calculate error bounds.

Syntax

```
call sla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
call dla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
call cla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
call zla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- `mkl.fi`

Description

The `?la_geamv` routines perform a matrix-vector operation defined as

$$y := \alpha * \text{abs}(A) * (x) + \beta * \text{abs}(y),$$

or

$$y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y),$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

trans CHARACTER*1. Specifies the operation:

if *trans* = `BLAS_NO_TRANS`, then $y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y)$

	<p>if $trans = \text{BLAS_TRANS}$, then $y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y)$</p> <p>if $trans = \text{'BLAS_CONJ_TRANS'}$, then $y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y)$.</p>
<i>m</i>	<p>INTEGER. Specifies the number of rows of the matrix <i>A</i>. The value of <i>m</i> must be at least zero.</p>
<i>n</i>	<p>INTEGER. Specifies the number of columns of the matrix <i>A</i>. The value of <i>n</i> must be at least zero.</p>
<i>alpha</i>	<p>REAL for <code>sla_geamv</code> and for <code>cla_geamv</code></p> <p>DOUBLE PRECISION for <code>dla_geamv</code> and <code>zla_geamv</code></p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <code>sla_geamv</code></p> <p>DOUBLE PRECISION for <code>dla_geamv</code></p> <p>COMPLEX for <code>cla_geamv</code></p> <p>DOUBLE COMPLEX for <code>zla_geamv</code></p> <p>Array, DIMENSION(<i>lda</i>, *). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix of coefficients. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.</p>
<i>x</i>	<p>REAL for <code>sla_geamv</code></p> <p>DOUBLE PRECISION for <code>dla_geamv</code></p> <p>COMPLEX for <code>cla_geamv</code></p> <p>DOUBLE COMPLEX for <code>zla_geamv</code></p> <p>Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$ when $trans = \text{'N'}$ or 'n' and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the incremented array <i>x</i> must contain the vector <i>X</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must be non-zero.</p>
<i>beta</i>	<p>REAL for <code>sla_geamv</code> and for <code>cla_geamv</code></p> <p>DOUBLE PRECISION for <code>dla_geamv</code> and <code>zla_geamv</code></p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is zero, you do not need to set <i>y</i> on input.</p>
<i>y</i>	<p>REAL for <code>sla_geamv</code> and for <code>cla_geamv</code></p> <p>DOUBLE PRECISION for <code>dla_geamv</code> and <code>zla_geamv</code></p> <p>Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when $trans = \text{'N'}$ or 'n' and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry with non-zero <i>beta</i>, the incremented array <i>y</i> must contain the vector <i>Y</i>.</p>

incy INTEGER. Specifies the increment for the elements of *y*.
The value of *incy* must be non-zero.

Output Parameters

y Updated vector *Y*.

?la_gercond

Estimates the Skeel condition number for a general matrix.

Syntax

```
call sla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

$op(A) * op2(C)$

where

the *cmode* parameter determines *op2* as follows:

<i>cmode</i> Value	op2(C)
1	<i>C</i>
0	<i>I</i>
-1	inv(<i>C</i>)

The Skeel condition number

$cond(A) = norminf(|inv(A)| |A|)$

is computed by computing scaling factors *R* such that

$diag(R) * A * op2(C)$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
Specifies the form of the system of equations:
If *trans* = 'N', the system has the form $A * X = B$ (No transpose).
If *trans* = 'T', the system has the form $A^T * X = B$ (Transpose).
If *trans* = 'C', the system has the form $A^H * X = B$ (Conjugate Transpose = Transpose).

<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i> , <i>af</i> , <i>c</i> , <i>work</i>	REAL for <code>sla_gercond</code> DOUBLE PRECISION for <code>dla_gercond</code> Arrays: <i>a</i> (<i>lda</i> ,*) contains the original general <i>n</i> -by- <i>n</i> matrix <i>A</i> . <i>af</i> (<i>ldaf</i> ,*) contains factors <i>L</i> and <i>U</i> from the factorization of the general matrix $A = P * L * U$, as returned by <code>?getrf</code> . <i>c</i> , DIMENSION <i>n</i> . The vector <i>C</i> in the formula $op(A) * op2(C)$. <i>work</i> is a workspace array of DIMENSION (3* <i>n</i>). The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> . $ldaf \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array with DIMENSION <i>n</i> . The pivot indices from the factorization $A = P * L * U$ as computed by <code>?getrf</code> . Row <i>i</i> of the matrix was interchanged with row <i>ipiv</i> (<i>i</i>).
<i>cmode</i>	INTEGER. Determines <i>op2</i> (<i>C</i>) in the formula $op(A) * op2(C)$ as follows: If <i>cmode</i> = 1, <i>op2</i> (<i>C</i>) = <i>C</i> . If <i>cmode</i> = 0, <i>op2</i> (<i>C</i>) = <i>I</i> . If <i>cmode</i> = -1, <i>op2</i> (<i>C</i>) = <i>inv</i> (<i>C</i>).
<i>iwork</i>	INTEGER. Workspace array with DIMENSION <i>n</i> .

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>i</i> > 0, the <i>i</i> -th parameter is invalid.
-------------	---

See Also

[?getrf](#)

[?la_gercond_c](#)

*Computes the infinity norm condition number of $op(A) * inv(diag(c))$ for general matrices.*

Syntax

```
call cla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- `mkl.fi`

Description

The function computes the infinity norm condition number of

$$\text{op}(A) * \text{inv}(\text{diag}(c))$$

where the `c` is a REAL vector for `cla_gercond_c` and a DOUBLE PRECISION vector for `zla_gercond_c`.

Input Parameters

<code>trans</code>	<p>CHARACTER*1. Must be 'N' or 'T' or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <code>trans = 'N'</code>, the system has the form $A*X = B$ (No transpose)</p> <p>If <code>trans = 'T'</code>, the system has the form $A^T*X = B$ (Transpose)</p> <p>If <code>trans = 'C'</code>, the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)</p>
<code>n</code>	<p>INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.</p>
<code>a, af, work</code>	<p>COMPLEX for <code>cla_gercond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_gercond_c</code></p> <p>Arrays:</p> <p><code>a(lda,*)</code> contains the original general n-by-n matrix A.</p> <p><code>af(ldaf,*)</code> contains the factors L and U from the factorization $A=P*L*U$ as returned by <code>?getrf</code>.</p> <p><code>work</code> is a workspace array of DIMENSION $(2*n)$.</p> <p>The second dimension of <code>a</code> and <code>af</code> must be at least $\max(1, n)$.</p>
<code>lda</code>	<p>INTEGER. The leading dimension of the array <code>a</code>. $lda \geq \max(1, n)$.</p>
<code>ldaf</code>	<p>INTEGER. The leading dimension of <code>af</code>. $ldaf \geq \max(1, n)$.</p>
<code>ipiv</code>	<p>INTEGER.</p> <p>Array with DIMENSION n. The pivot indices from the factorization $A = P*L*U$ as computed by <code>?getrf</code>. Row i of the matrix was interchanged with row <code>ipiv(i)</code>.</p>
<code>c, rwork</code>	<p>REAL for <code>cla_gercond_c</code></p> <p>DOUBLE PRECISION for <code>zla_gercond_c</code></p> <p>Array <code>c</code> with DIMENSION n. The vector <code>c</code> in the formula</p> $\text{op}(A) * \text{inv}(\text{diag}(c)).$ <p>Array <code>rwork</code> with DIMENSION n is a workspace.</p>
<code>capply</code>	<p>LOGICAL. If <code>capply=.TRUE.</code>, then the function uses the vector <code>c</code> from the formula</p>

$op(A) * inv(diag(c)).$

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

?getrf

?la_gercond_x

*Computes the infinity norm condition number of $op(A)*diag(x)$ for general matrices.*

Syntax

call cla_gercond_x(*trans*, *n*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *x*, *info*, *work*, *rwork*)

call zla_gercond_x(*trans*, *n*, *a*, *lda*, *af*, *ldaf*, *ipiv*, *x*, *info*, *work*, *rwork*)

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$op(A) * diag(x)$

where the *x* is a COMPLEX vector for cla_gercond_x and a DOUBLE COMPLEX vector for zla_gercond_x.

Input Parameters

trans CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Specifies the form of the system of equations:
 If *trans* = 'N', the system has the form $A*X = B$ (No transpose)
 If *trans* = 'T', the system has the form $A^T*X = B$ (Transpose)
 If *trans* = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a, *af*, *x*, *work* COMPLEX for cla_gercond_x
 DOUBLE COMPLEX for zla_gercond_x
 Arrays:
a(lda,)* contains the original general *n*-by-*n* matrix *A*.
af(ldaf,)* contains the factors *L* and *U* from the factorization $A=P*L*U$ as returned by ?getrf.
x, DIMENSION*n*. The vector *x* in the formula $op(A) * diag(x)$.

work is a workspace array of DIMENSION (2*n).

The second dimension of *a* and *af* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*. $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array with DIMENSION *n*. The pivot indices from the factorization $A = P*L*U$ as computed by ?getrf. Row *i* of the matrix was interchanged with row *ipiv(i)*.

rwork REAL for cla_gercond_x
 DOUBLE PRECISION for zla_gercond_x
 Array *rwork* with DIMENSION *n* is a workspace.

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

[?getrf](#)

[?la_gerfsx_extended](#)

Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call sla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
  colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
  rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call dla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
  colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
  rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call cla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
  colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
  rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call zla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
  colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
  rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- mkl.fi

Description

The `?la_gerfsx_extended` subroutine improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?gerfsx` routine calls `?la_gerfsx_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `errs_n` and `errs_c` for details of the error bounds.

Use `?la_gerfsx_extended` to set only the second fields of `errs_n` and `errs_c`.

Input Parameters

<code>prec_type</code>	<p>INTEGER.</p> <p>Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code>, where <code>p</code> is a CHARACTER and:</p> <p>If <code>p = 'S'</code>: Single.</p> <p>If <code>p = 'D'</code>: Double.</p> <p>If <code>p = 'I'</code>: Indigenous.</p> <p>If <code>p = 'X', 'E'</code>: Extra.</p>
<code>trans_type</code>	<p>INTEGER.</p> <p>Specifies the transposition operation on <i>A</i>. The value is defined by <code>ilatrans(t)</code>, where <code>t</code> is a CHARACTER and:</p> <p>If <code>t = 'N'</code>: No transpose.</p> <p>If <code>t = 'T'</code>: Transpose.</p> <p>If <code>t = 'C'</code>: Conjugate Transpose.</p>
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrix <i>B</i> .
<code>a, af, b, y</code>	<p>REAL for <code>sla_gerfsx_extended</code></p> <p>DOUBLE PRECISION for <code>dla_gerfsx_extended</code></p> <p>COMPLEX for <code>cla_gerfsx_extended</code></p> <p>DOUBLE COMPLEX for <code>zla_gerfsx_extended</code>.</p> <p>Arrays: <code>a(lda,*)</code>, <code>af(ldaf,*)</code>, <code>b(ldb,*)</code>, <code>y(ldy,*)</code>.</p> <p>The array <code>a</code> contains the original matrix <i>n</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <code>a</code> must be at least <code>max(1, n)</code>.</p> <p>The array <code>af</code> contains the factors <i>L</i> and <i>U</i> from the factorization $A = P * L * U$ as computed by <code>?getrf</code>. The second dimension of <code>af</code> must be at least <code>max(1, n)</code>.</p> <p>The array <code>b</code> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <code>b</code> must be at least <code>max(1, nrhs)</code>.</p>

The array y on entry contains the solution matrix X as computed by ?getrs. The second dimension of y must be at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of the array af ; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices from the factorization $A = P*L*U$ as computed by ?getrf; row i of the matrix was interchanged with row $ipiv(i)$.

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to A before calling this routine. This is needed to compute the solution and error bounds correctly.

c REAL for single precision flavors (sla_gersx_extended, cla_gersx_extended)
DOUBLE PRECISION for double precision flavors (dla_gersx_extended, zla_gersx_extended).

c contains the column scale factors for A . If *colequ* = .FALSE., c is not used.

If c is input, each element of c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

ldy INTEGER. The leading dimension of the array y ; $ldy \geq \max(1, n)$.

n_norms INTEGER. Determines which error bounds to return. See *errs_n* and *errs_c* descriptions in *Output Arguments* section below.

If $n_norms \geq 1$, returns normwise error bounds.

If $n_norms \geq 2$, returns componentwise error bounds.

errs_n REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION($nrhs, n_err_bnds$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in $errs_n(i, :)$ corresponds to the i -th right-hand side.

The second index in $errs_n(:, err)$ contains the following three fields:

$err=1$	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
$err=2$	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
$err=3$	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.

$errs_c$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION($nrhs, n_err_bnds$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for

each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then $errs_c$ is not accessed. If $n_err_bnds < 3$, then at most the first $(:,n_err_bnds)$ entries are returned.

The first index in $errs_c(i,:)$ corresponds to the i -th right-hand side.

The second index in $errs_c(:,err)$ contains the following three fields:

$err=1$	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
$err=2$	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
$err=3$	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.

res, dy, y_tail

REAL for `sla_gerfsx_extended`
 DOUBLE PRECISION for `dla_gerfsx_extended`
 COMPLEX for `cla_gerfsx_extended`
 DOUBLE COMPLEX for `zla_gerfsx_extended`.

Workspace arrays of DIMENSION n .

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION n .

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *errs_n* and *errs_c* may no longer be trustworthy.

rthresh

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$$\text{norm}(\text{dx}_{\{i+1\}}) < \text{rthresh} * \text{norm}(\text{dx}_i)$$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$$0 < \text{rthresh} \leq 1.$$

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

REAL for `sla_gerfsx_extended`

DOUBLE PRECISION for `dla_gerfsx_extended`

COMPLEX for `cla_gerfsx_extended`

DOUBLE COMPLEX for `zla_gerfsx_extended`.

	The improved solution matrix Y .
<code>berr_out</code>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for right-hand-side j from the formula</p> $\max(i) \left(\frac{\text{abs}(\text{res}(i))}{\left(\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B) \right) (i)} \right)$ <p>where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z. This is computed by <code>?la_lin_berr</code>.</p>
<code>errs_n, errs_c</code>	<p>Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array $(1:nrhs, 2)$. The other elements are kept unchanged.</p>
<code>info</code>	<p>INTEGER. If <code>info = 0</code>, the execution is successful. The solution to every right-hand side is guaranteed.</p> <p>If <code>info = -i</code>, the i-th parameter had an illegal value.</p>

See Also

[?gerfsx](#)
[?getrf](#)
[sgetrs dgetrs cgetrs zgetrs Reference](#)
[?lamch](#)
[ilaprec](#)
[ilatrans](#)
[?la_lin_berr](#)

?la_heamv

Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.

Syntax

```
call cla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- `mkl.fi`

Description

The `?la_heamv` routines perform a matrix-vector operation defined as

$$y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y),$$

where:

α and β are scalars,

x and y are vectors,

A is an n -by- n Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array A is to be referenced: If <i>uplo</i> = 'BLAS_UPPER', only the upper triangular part of A is to be referenced, If <i>uplo</i> = 'BLAS_LOWER', only the lower triangular part of A is to be referenced.
<i>n</i>	INTEGER. Specifies the number of rows and columns of the matrix A. The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <i>cla_heamv</i> DOUBLE PRECISION for <i>zla_heamv</i> Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <i>cla_heamv</i> DOUBLE COMPLEX for <i>zla_heamv</i> Array, DIMENSION(<i>lda</i> , *). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	COMPLEX for <i>cla_heamv</i> DOUBLE COMPLEX for <i>zla_heamv</i> Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the vector <i>X</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must be non-zero.
<i>beta</i>	REAL for <i>cla_heamv</i> DOUBLE PRECISION for <i>zla_heamv</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is zero, you do not need to set <i>y</i> on input.
<i>y</i>	REAL for <i>cla_heamv</i> DOUBLE PRECISION for <i>zla_heamv</i>

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *Y*.

incy INTEGER. Specifies the increment for the elements of *y*.
The value of *incy* must be non-zero.

Output Parameters

y Updated vector *Y*.

?la_hercond_c

*Computes the infinity norm condition number of $op(A) * inv(diag(c))$ for Hermitian indefinite matrices.*

Syntax

```
call cla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$op(A) * inv(diag(c))$

where the *c* is a REAL vector for `cla_hercond_c` and a DOUBLE PRECISION vector for `zla_hercond_c`.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Specifies the triangle of *A* to store:
If *uplo* = 'U', the upper triangle of *A* is stored,
If *uplo* = 'L', the lower triangle of *A* is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix *A*; $n \geq 0$.

a COMPLEX for `cla_hercond_c`
DOUBLE COMPLEX for `zla_hercond_c`
Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix *A*. The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

af COMPLEX for `cla_hercond_c`
DOUBLE COMPLEX for `zla_hercond_c`

Array, DIMENSION($ldaf$, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by `?hetrf`. The second dimension of af must be at least $\max(1, n)$.

ldaf INTEGER. The leading dimension of the array *af*. $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by `?hetrf`.

c REAL for `cla_hercond_c`
DOUBLE PRECISION for `zla_hercond_c`

Array *c* with DIMENSION n . The vector *c* in the formula
 $\text{op}(A) * \text{inv}(\text{diag}(c))$.

caply LOGICAL. If `.TRUE.`, then the function uses the vector *c* from the formula
 $\text{op}(A) * \text{inv}(\text{diag}(c))$.

work COMPLEX for `cla_hercond_c`
DOUBLE COMPLEX for `zla_hercond_c`

Array DIMENSION $2*n$. Workspace.

rwork REAL for `cla_hercond_c`
DOUBLE PRECISION for `zla_hercond_c`

Array DIMENSION n . Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

`?hetrf`

`?la_hercond_x`

Computes the infinity norm condition number of $\text{op}(A)\text{diag}(x)$ for Hermitian indefinite matrices.*

Syntax

```
call cla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

```
call zla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- `mkl.fi`

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_hercond_x` and a DOUBLE COMPLEX vector for `zla_hercond_x`.

Input Parameters

<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Specifies the triangle of A to store:</p> <p>If <code>uplo = 'U'</code>, the upper triangle of A is stored,</p> <p>If <code>uplo = 'L'</code>, the lower triangle of A is stored.</p>
<code>n</code>	<p>INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.</p>
<code>a</code>	<p>COMPLEX for <code>cla_hercond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_hercond_c</code></p> <p>Array, DIMENSION(<code>lda</code>, *). On entry, the n-by-n matrix A. The second dimension of <code>a</code> must be at least $\max(1, n)$.</p>
<code>lda</code>	<p>INTEGER. The leading dimension of the array <code>a</code>. $lda \geq \max(1, n)$.</p>
<code>af</code>	<p>COMPLEX for <code>cla_hercond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_hercond_c</code></p> <p>Array, DIMENSION(<code>ldaf</code>, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?hetrf</code>. The second dimension of <code>af</code> must be at least $\max(1, n)$.</p>
<code>ldaf</code>	<p>INTEGER. The leading dimension of the array <code>af</code>. $ldaf \geq \max(1, n)$.</p>
<code>ipiv</code>	<p>INTEGER.</p> <p>Array with DIMENSION n. Details of the interchanges and the block structure of D as determined by <code>?hetrf</code>.</p>
<code>x</code>	<p>COMPLEX for <code>cla_hercond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_hercond_c</code></p> <p>Array <code>x</code> with DIMENSION n. The vector x in the formula $\text{op}(A) * \text{inv}(\text{diag}(x))$.</p>
<code>work</code>	<p>COMPLEX for <code>cla_hercond_c</code></p> <p>DOUBLE COMPLEX for <code>zla_hercond_c</code></p> <p>Array DIMENSION $2*n$. Workspace.</p>
<code>rwork</code>	<p>REAL for <code>cla_hercond_c</code></p> <p>DOUBLE PRECISION for <code>zla_hercond_c</code></p> <p>Array DIMENSION n. Workspace.</p>

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

?hetrf

?la_herfsx_extended

Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call cla_herfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call zla_herfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- mkl.fi

Description

The ?la_herfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?herfsx routine calls ?la_herfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_herfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

prec_type INTEGER.
 Specifies the intermediate precision to be used in refinement. The value is defined by *ilaprec(p)*, where *p* is a CHARACTER and:
 If *p* = 'S': Single.
 If *p* = 'D': Double.
 If *p* = 'I': Indigenous.
 If *p* = 'X', 'E': Extra.

uplo CHARACTER*1. Must be 'U' or 'L'.
 Specifies the triangle of A to store:

	<p>If <code>uplo = 'U'</code>, the upper triangle of A is stored,</p> <p>If <code>uplo = 'L'</code>, the lower triangle of A is stored.</p>
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrix B .
<code>a, af, b, y</code>	<p>COMPLEX for <code>cla_herfsx_extended</code></p> <p>DOUBLE COMPLEX for <code>zla_herfsx_extended</code>.</p> <p>Arrays: <code>a(lda,*)</code>, <code>af(ldaf,*)</code>, <code>b ldb,*)</code>, <code>y(ldy,*)</code>.</p> <p>The array <code>a</code> contains the original n-by-n matrix A. The second dimension of <code>a</code> must be at least $\max(1, n)$.</p> <p>The array <code>af</code> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?hetrf</code>. The second dimension of <code>af</code> must be at least $\max(1, n)$.</p> <p>The array <code>b</code> contains the right-hand-side of the matrix B. The second dimension of <code>b</code> must be at least $\max(1, nrhs)$.</p> <p>The array <code>y</code> on entry contains the solution matrix X as computed by <code>?hetrs</code>. The second dimension of <code>y</code> must be at least $\max(1, nrhs)$.</p>
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> ; $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> ; $ldaf \geq \max(1, n)$.
<code>ipiv</code>	<p>INTEGER.</p> <p>Array, DIMENSION n. Details of the interchanges and the block structure of D as determined by <code>?hetrf</code>.</p>
<code>colequ</code>	LOGICAL. If <code>colequ = .TRUE.</code> , column equilibration was done to A before calling this routine. This is needed to compute the solution and error bounds correctly.
<code>c</code>	<p>REAL for <code>cla_herfsx_extended</code></p> <p>DOUBLE PRECISION for <code>zla_herfsx_extended</code>.</p> <p><code>c</code> contains the column scale factors for A. If <code>colequ = .FALSE.</code>, <code>c</code> is not used.</p> <p>If <code>c</code> is input, each element of <code>c</code> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>
<code>ldb</code>	INTEGER. The leading dimension of the array <code>b</code> ; $ldb \geq \max(1, n)$.
<code>ldy</code>	INTEGER. The leading dimension of the array <code>y</code> ; $ldy \geq \max(1, n)$.
<code>n_norms</code>	INTEGER. Determines which error bounds to return. See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below.

If $n_norms \geq 1$, returns normwise error bounds.

If $n_norms \geq 2$, returns componentwise error bounds.

err_bnds_norm

REAL for *cla_herfsx_extended*

DOUBLE PRECISION for *zla_herfsx_extended*.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i, :)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:, err)* contains the following three fields:

- | | |
|--------------|---|
| <i>err=1</i> | "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>cla_herfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>zla_herfsx_extended</i> . |
| <i>err=2</i> | "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>cla_herfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>zla_herfsx_extended</i> . This error bound should only be trusted if the previous boolean is true. |
| <i>err=3</i> | Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>cla_herfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>zla_herfsx_extended</i> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix <i>Z</i> . |

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`err_bnds_comp`

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Array, DIMENSION(`nrhs`, `n_err_bnds`). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first (`:, n_err_bnds`) entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

- `err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `cla_herfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zla_herfsx_extended`.
- `err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `cla_herfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zla_herfsx_extended`. This error bound should only be trusted if the previous boolean is true.
- `err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for

`cla_herfsx_extended` and `sqrt(n)*dlamch(ε)` for `zla_herfsx_extended` to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1/(\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`res, dy, y_tail`

COMPLEX for `cla_herfsx_extended`
DOUBLE COMPLEX for `zla_herfsx_extended`.

Workspace arrays of DIMENSION n .

`res` holds the intermediate residual.

`dy` holds the intermediate solution.

`y_tail` holds the trailing bits of the intermediate solution.

`ayb`

REAL for `cla_herfsx_extended`
DOUBLE PRECISION for `zla_herfsx_extended`.

Workspace array, DIMENSION n .

`rcond`

REAL for `cla_herfsx_extended`
DOUBLE PRECISION for `zla_herfsx_extended`.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond` = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

`ithresh`

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`rthresh`

REAL for `cla_herfsx_extended`
DOUBLE PRECISION for `zla_herfsx_extended`.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(dx_{\{i+1\}}) < rthresh * \text{norm}(dx_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$$0 < rthresh \leq 1.$$

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution *y* is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

COMPLEX for `cla_herfsx_extended`

DOUBLE COMPLEX for `zla_herfsx_extended`.

The improved solution matrix *Y*.

berr_out

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Array, DIMENSION *nrhs*. *berr_out*(*j*) contains the componentwise relative backward error for right-hand-side *j* from the formula

$$\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B)) (i))$$

where *abs*(*z*) is the componentwise absolute value of the matrix or vector *Z*. This is computed by `?la_lin_berr`.

err_bnds_norm,
err_bnds_comp

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (`1:nrhs, 2`). The other elements are kept unchanged.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

See Also

[?herfsx](#)

[?hetrf](#)

[?hetrs](#)

[?lamch](#)

[ilaprec](#)

[ilatrans](#)

[?la_lin_berr](#)

?la_herpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a Hermitian indefinite matrix.

Syntax

```
call cla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

```
call zla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- mkl.fi

Description

The ?la_herpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix A could be poor. This also means that the solution X, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <i>uplo</i> = 'U', the upper triangle of A is stored, If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix A; $n \geq 0$.
<i>info</i>	INTEGER. The value of INFO returned from ?hetrf, that is, the pivot in column <i>info</i> is exactly 0.
<i>a, af</i>	COMPLEX for cla_herpvgrw DOUBLE COMPLEX for zla_herpvgrw. Arrays: <i>a</i> (<i>lda</i> ,*), <i>af</i> (<i>ldaf</i> ,*). <i>a</i> contains the <i>n</i> -by- <i>n</i> matrix A. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>af</i> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf. The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of array <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION <i>n</i> . Details of the interchanges and the block structure of D as determined by ?hetrf.
<i>work</i>	REAL for cla_herpvgrw

DOUBLE PRECISION for `zla_herpvgrw`.

Array, DIMENSION $2*n$. Workspace.

See Also

[?hetrf](#)

[?la_lin_berr](#)

Computes component-wise relative backward error.

Syntax

```
call sla_lin_berr(n, nz, nrhs, res, ayb, berr )
```

```
call dla_lin_berr(n, nz, nrhs, res, ayb, berr )
```

```
call cla_lin_berr(n, nz, nrhs, res, ayb, berr )
```

```
call zla_lin_berr(n, nz, nrhs, res, ayb, berr )
```

Include Files

- `mkl.fi`

Description

The `?la_lin_berr` computes a component-wise relative backward error from the formula:

$$\max(i) \left(\frac{\text{abs}(R(i))}{(\text{abs}(\text{op}(A_s)) * \text{abs}(Y) + \text{abs}(B_s)) (i)} \right)$$

where $\text{abs}(Z)$ is the component-wise value of the matrix or vector Z .

Input Parameters

<code>n</code>	INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.
<code>nz</code>	INTEGER. The parameter for guarding against spuriously zero residuals. $(nz+1) * \text{slamch}(\text{'Safe minimum'})$ is added to $R(i)$ in the numerator of the relative backward error formula. The default value is n .
<code>nrhs</code>	INTEGER. Number of right-hand sides, the number of columns in the matrices AYB , RES , and $BERR$; $nrhs \geq 0$.
<code>res, ayb</code>	REAL for <code>sla_lin_berr</code> , <code>cla_lin_berr</code> DOUBLE PRECISION for <code>dla_lin_berr</code> , <code>zla_lin_berr</code> Arrays, DIMENSION $(n, nrhs)$. <code>res</code> is the residual matrix, that is, the matrix R in the relative backward error formula. <code>ayb</code> is the denominator of that formula, that is, the matrix $\text{abs}(\text{op}(A_s)) * \text{abs}(Y) + \text{abs}(B_s)$. The matrices A , Y , and B are from iterative refinement. See description of <code>?la_gersfx_extended</code> .

Output Parameters

`berr` REAL for `sla_lin_berr`

DOUBLE PRECISION for `dla_lin_berr`
 COMPLEX for `cla_lin_berr`
 DOUBLE COMPLEX for `zla_lin_berr`
 The component-wise relative backward error.

See Also

[?lamch](#)
[?la_gerfsx_extended](#)

?la_porcond

Estimates the Skeel condition number for a symmetric positive-definite matrix.

Syntax

```
call sla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
call dla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
```

Include Files

- `mkl.fi`

Description

The function estimates the Skeel condition number of

$op(A) * op2(C)$

where

the `cmode` parameter determines `op2` as follows:

<i>cmode</i> Value	op2(C)
1	C
0	I
-1	$inv(C)$

The Skeel condition number

$cond(A) = norminf(|inv(A)| |A|)$

is computed by computing scaling factors R such that

$diag(R) * A * op2(C)$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Specifies the triangle of A to store:
 If *uplo* = 'U', the upper triangle of A is stored,
 If *uplo* = 'L', the lower triangle of A is stored.

<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i> , <i>af</i> , <i>c</i> , <i>work</i>	REAL for <code>sla_porcond</code> DOUBLE PRECISION for <code>dla_porcond</code> Arrays: <i>a</i> (<i>lda</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix <i>A</i> . <i>af</i> (<i>ldaf</i> ,*) contains the triangular factor <i>L</i> or <i>U</i> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$, as computed by <code>?potrf</code> . <i>c</i> , DIMENSION <i>n</i> . The vector <i>C</i> in the formula $op(A) * op2(C)$. <i>work</i> is a workspace array of DIMENSION (3* <i>n</i>). The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>ab</i> . $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> . $ldaf \geq \max(1, n)$.
<i>cmode</i>	INTEGER. Determines $op2(C)$ in the formula $op(A) * op2(C)$ as follows: If <i>cmode</i> = 1, $op2(C) = C$. If <i>cmode</i> = 0, $op2(C) = I$. If <i>cmode</i> = -1, $op2(C) = inv(C)$.
<i>iwork</i>	INTEGER. Workspace array with DIMENSION <i>n</i> .

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>i</i> > 0, the <i>i</i> -th parameter is invalid.
-------------	---

See Also

[?potrf](#)

[?la_porcond_c](#)

*Computes the infinity norm condition number of $op(A) * inv(diag(c))$ for Hermitian positive-definite matrices.*

Syntax

```
call cla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
call zla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
```

Include Files

- `mkl.fi`

Description

The function computes the infinity norm condition number of

$$\text{op}(A) * \text{inv}(\text{diag}(c))$$

where the c is a REAL vector for `cla_porcond_c` and a DOUBLE PRECISION vector for `zla_porcond_c`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <code>uplo</code> = 'U', the upper triangle of A is stored, If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>a</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array, DIMENSION(<code>lda</code> , *). On entry, the n -by- n matrix A . The second dimension of <code>a</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $lda \geq \max(1, n)$.
<code>af</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array, DIMENSION(<code>ldaf</code> , *). The triangular factor L or U from the Cholesky factorization $A = U^H * U$ or $A = L * L^H$, as computed by <code>?potrf</code> . The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> . $ldaf \geq \max(1, n)$.
<code>c</code>	REAL for <code>cla_porcond_c</code> DOUBLE PRECISION for <code>zla_porcond_c</code> Array <code>c</code> with DIMENSION n . The vector c in the formula $\text{op}(A) * \text{inv}(\text{diag}(c))$.
<code>capply</code>	LOGICAL. If <code>.TRUE.</code> , then the function uses the vector <code>c</code> from the formula $\text{op}(A) * \text{inv}(\text{diag}(c))$.
<code>work</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array DIMENSION $2*n$. Workspace.
<code>rwork</code>	REAL for <code>cla_porcond_c</code> DOUBLE PRECISION for <code>zla_porcond_c</code>

Array DIMENSION n . Workspace.

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

?potrf

?la_porcond_x

*Computes the infinity norm condition number of $op(A)*diag(x)$ for Hermitian positive-definite matrices.*

Syntax

```
call cla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
call zla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$op(A) * diag(x)$

where the *x* is a COMPLEX vector for cla_porcond_x and a DOUBLE COMPLEX vector for zla_porcond_x.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
 Specifies the triangle of A to store:
 If *uplo* = 'U', the upper triangle of A is stored,
 If *uplo* = 'L', the lower triangle of A is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.

a COMPLEX for cla_porcond_c
 DOUBLE COMPLEX for zla_porcond_c
 Array, DIMENSION(*lda*, *). On entry, the *n*-by-*n* matrix A.
 The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

af COMPLEX for cla_porcond_c
 DOUBLE COMPLEX for zla_porcond_c

Array, DIMENSION(*ldaf*, *). The triangular factor *L* or *U* from the Cholesky factorization

$$A = U^H * U \text{ or } A = L * L^H,$$

as computed by `?potrf`.

The second dimension of *af* must be at least $\max(1, n)$.

ldaf INTEGER. The leading dimension of the array *af*. $ldaf \geq \max(1, n)$.

x COMPLEX for `cla_porcond_c`

DOUBLE COMPLEX for `zla_porcond_c`

Array *x* with DIMENSION *n*. The vector *x* in the formula

$$\text{op}(A) * \text{inv}(\text{diag}(x)).$$

work COMPLEX for `cla_porcond_c`

DOUBLE COMPLEX for `zla_porcond_c`

Array DIMENSION $2 * n$. Workspace.

rwork REAL for `cla_porcond_c`

DOUBLE PRECISION for `zla_porcond_c`

Array DIMENSION *n*. Workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If *i* > 0, the *i*-th parameter is invalid.

See Also

[?potrf](#)

?la_porfsx_extended

Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call sla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call dla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call cla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

call `zla_porfsx_extended(prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise, info)`

Include Files

- `mkl.fi`

Description

The `?la_porfsx_extended` subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?herfsx` routine calls `?la_porfsx_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use `?la_porfsx_extended` to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

<code>prec_type</code>	INTEGER. Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code> , where <code>p</code> is a CHARACTER and: If <code>p = 'S'</code> : Single. If <code>p = 'D'</code> : Double. If <code>p = 'I'</code> : Indigenous. If <code>p = 'X', 'E'</code> : Extra.
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of <code>A</code> to store: If <code>uplo = 'U'</code> , the upper triangle of <code>A</code> is stored, If <code>uplo = 'L'</code> , the lower triangle of <code>A</code> is stored.
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix <code>A</code> ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrix <code>B</code> .
<code>a, af, b, y</code>	REAL for <code>sla_porfsx_extended</code> DOUBLE PRECISION for <code>dla_porfsx_extended</code> COMPLEX for <code>cla_porfsx_extended</code> DOUBLE COMPLEX for <code>zla_porfsx_extended</code> . Arrays: <code>a(lda,*)</code> , <code>af(ldaf,*)</code> , <code>b(ldb,*)</code> , <code>y(ldy,*)</code> . The array <code>a</code> contains the original n -by- n matrix <code>A</code> . The second dimension of <code>a</code> must be at least <code>max(1, n)</code> . The array <code>af</code> contains the triangular factor <code>L</code> or <code>U</code> from the Cholesky factorization as computed by <code>?potrf</code> : $A = U^T * U$ or $A = L * L^T$ for real flavors, $A = U^H * U$ or $A = L * L^H$ for complex flavors.

The second dimension of a must be at least $\max(1, n)$.

The array b contains the right-hand-side of the matrix B . The second dimension of b must be at least $\max(1, nrhs)$.

The array y on entry contains the solution matrix X as computed by `potrs`. The second dimension of y must be at least $\max(1, nrhs)$.

<code>lda</code>	INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array a ; $ldaf \geq \max(1, n)$.
<code>colequ</code>	LOGICAL. If <code>colequ = .TRUE.</code> , column equilibration was done to A before calling this routine. This is needed to compute the solution and error bounds correctly.
<code>c</code>	REAL for <code>sla_porfsx_extended</code> and <code>cla_porfsx_extended</code> DOUBLE PRECISION for <code>dla_porfsx_extended</code> and <code>zla_porfsx_extended</code> . c contains the column scale factors for A . If <code>colequ = .FALSE.</code> , c is not used. If c is input, each element of c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.
<code>ldb</code>	INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.
<code>ldy</code>	INTEGER. The leading dimension of the array y ; $ldy \geq \max(1, n)$.
<code>n_norms</code>	INTEGER. Determines which error bounds to return. See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below. If $n_norms \geq 1$, returns normwise error bounds. If $n_norms \geq 2$, returns componentwise error bounds.
<code>err_bnds_norm</code>	REAL for <code>sla_porfsx_extended</code> and <code>cla_porfsx_extended</code> DOUBLE PRECISION for <code>dla_porfsx_extended</code> and <code>zla_porfsx_extended</code> . Array, DIMENSION($nrhs, n_err_bnds$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error. Normwise relative error in the i -th solution vector is defined as follows: $\frac{\max_j X_{true_{ji}} - X_{ji} }{\max_j X_{ji} }$ The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

<code>err=1</code>	<p>"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code>.</p>
<code>err=2</code>	<p>"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code>. This error bound should only be trusted if the previous boolean is true.</p>
<code>err=3</code>	<p>Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix <i>Z</i>.</p> <p>Let $z = s * a$, where <i>s</i> scales each row by a power of the radix so all absolute row sums of <i>z</i> are approximately 1.</p> <p>Use this subroutine to set only the second field above.</p>

`err_bnds_comp`

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and `zla_porfsx_extended`.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params*(3) = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first (*:,n_err_bnds*) entries are returned.

The first index in *err_bnds_comp*(*i*,*:*) corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp*(*:*,*err*) contains the following three fields:

- | | |
|---------------|--|
| <i>err</i> =1 | "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>sla_porfsx_extended</i> / <i>cla_porfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>dla_porfsx_extended</i> / <i>zla_porfsx_extended</i> . |
| <i>err</i> =2 | "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>sla_porfsx_extended</i> / <i>cla_porfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>dla_porfsx_extended</i> / <i>zla_porfsx_extended</i> . This error bound should only be trusted if the previous boolean is true. |
| <i>err</i> =3 | Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <i>sla_porfsx_extended</i> / <i>cla_porfsx_extended</i> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <i>dla_porfsx_extended</i> / <i>zla_porfsx_extended</i> to determine if the error estimate is "guaranteed". These reciprocal |

condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

res, dy, y_tail

REAL for `sla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended`
 COMPLEX for `cla_porfsx_extended`
 DOUBLE COMPLEX for `zla_porfsx_extended`.

Workspace arrays of DIMENSION n .

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Workspace array, DIMENSION n .

rcond

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(dx_{\{i+1\}}) < rthresh * \text{norm}(dx_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$0 < rthresh \leq 1$.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for *sla_porfsx_extended* and *cla_porfsx_extended*

DOUBLE PRECISION for *dla_porfsx_extended* and
zla_porfsx_extended.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

REAL for *sla_porfsx_extended*

DOUBLE PRECISION for *dla_porfsx_extended*

COMPLEX for *cla_porfsx_extended*

DOUBLE COMPLEX for *zla_porfsx_extended*.

The improved solution matrix Y .

berr_out

REAL for *sla_porfsx_extended* and *cla_porfsx_extended*

DOUBLE PRECISION for *dla_porfsx_extended* and
zla_porfsx_extended.

Array, DIMENSION *nrhs*. *berr_out*(*j*) contains the componentwise relative backward error for right-hand-side *j* from the formula

$$\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B)) (i))$$

where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z . This is computed by `?la_lin_berr`.

err_bnds_norm,
err_bnds_comp

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (`1:nrhs, 2`). The other elements are kept unchanged.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = *-i*, the *i*-th parameter had an illegal value.

See Also

[?porfsx](#)

[?potrf](#)

?potrs
 ?lamch
 ilaprec
 ilatrans
 ?la_lin_berr

?la_porpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a symmetric or Hermitian positive-definite matrix.

Syntax

```
call sla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call dla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call cla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call zla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
```

Include Files

- mkl.fi

Description

The ?la_porpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Specifies the triangle of <i>A</i> to store:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored,</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>ncols</i>	<p>INTEGER. The number of columns of the matrix <i>A</i>; <i>ncols</i> ≥ 0.</p>
<i>a, af</i>	<p>REAL for sla_porpvgrw</p> <p>DOUBLE PRECISION for dla_porpvgrw</p> <p>COMPLEX for cla_porpvgrw</p> <p>DOUBLE COMPLEX for zla_porpvgrw.</p> <p>Arrays: <i>a(lda,*)</i>, <i>af(ldaf,*)</i>.</p> <p>The array <i>a</i> contains the input <i>n</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p>The array <i>af</i> contains the triangular factor <i>L</i> or <i>U</i> from the Cholesky factorization as computed by ?potrf:</p> <p>$A = U^T * U$ or $A = L * L^T$ for real flavors,</p> <p>$A = U^H * U$ or $A = L * L^H$ for complex flavors.</p>

	The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>work</i>	REAL for <i>sla_porpvgrw</i> and <i>cla_porpvgrw</i> DOUBLE PRECISION for <i>dla_porpvgrw</i> and <i>zla_porpvgrw</i> . Workspace array, dimension $2*n$.

See Also

?potrf

?laqhe

Scales a Hermitian matrix.

Syntax

```
call claqhe( uplo, n, a, lda, s, scond, amax, equed )
```

```
call zlaqhe( uplo, n, a, lda, s, scond, amax, equed )
```

Include Files

- mkl.fi

Description

The routine equilibrates a Hermitian matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether to store the upper or lower part of the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', the upper triangular part of <i>A</i> ; if <i>uplo</i> = 'L', the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	COMPLEX for <i>claqhe</i> DOUBLE COMPLEX for <i>zlaqhe</i> Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> .

$lda \geq \max(n, 1)$.

s REAL for `claqhe`
 DOUBLE PRECISION for `zlaqhe`
 Array, DIMENSION (*n*). The scale factors for *A*.

scond REAL for `claqhe`
 DOUBLE PRECISION for `zlaqhe`
 Ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax REAL for `claqhe`
 DOUBLE PRECISION for `zlaqhe`
 Absolute value of largest matrix entry.

Output Parameters

a If *equed* = 'Y', *a* contains the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$.

equed CHARACTER*1.
 Specifies whether or not equilibration was done.
 If *equed* = 'N': No equilibration.
 If *equed* = 'Y': Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If $scond < thresh$, scaling is done.

The *large* and *small* parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqhp

Scales a Hermitian matrix stored in packed form.

Syntax

```
call claqhp( uplo, n, ap, s, scnd, amax, equed )
```

```
call zlaqhp( uplo, n, ap, s, scnd, amax, equed )
```

Include Files

- `mkl.fi`

Description

The routine equilibrates a Hermitian matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether to store the upper or lower part of the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', the upper triangular part of <i>A</i> ; if <i>uplo</i> = 'L', the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>ap</i>	COMPLEX for <i>claqhp</i> DOUBLE COMPLEX for <i>zlaqhp</i> Array, DIMENSION $(n*(n+1)/2)$. The Hermitian matrix <i>A</i> . <ul style="list-style-type: none"> If <i>uplo</i> = 'U', the upper triangular part of the Hermitian matrix <i>A</i> is stored in the packed array <i>ap</i> as follows: $ap(i+(j-1)*j/2) = A(i, j)$ for $1 \leq i \leq j$. If <i>uplo</i> = 'L', the lower triangular part of Hermitian matrix <i>A</i> is stored in the packed array <i>ap</i> as follows: $ap(i+(j-1)*(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.
<i>s</i>	REAL for <i>claqhp</i> DOUBLE PRECISION for <i>zlaqhp</i> Array, DIMENSION (n) . The scale factors for <i>A</i> .
<i>scond</i>	REAL for <i>claqhp</i> DOUBLE PRECISION for <i>zlaqhp</i> Ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for <i>claqhp</i> DOUBLE PRECISION for <i>zlaqhp</i> Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	If <i>equed</i> = 'Y', <i>a</i> contains the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$ in the same storage format as on input.
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If <i>equed</i> = 'N': No equilibration. If <i>equed</i> = 'Y': Equilibration was done, that is, <i>A</i> has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If $scond < thresh$, scaling is done.

The *large* and *small* parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?larcm

Multiplies a square real matrix by a complex matrix.

Syntax

```
call clarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

```
call zlarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- mkl.fi

Description

The routine performs a simple matrix-matrix multiplication of the form

$$C = A*B,$$

where *A* is *m*-by-*m* and real, *B* is *m*-by-*n* and complex, *C* is *m*-by-*n* and complex.

Input Parameters

<i>m</i>	INTEGER. The number of rows and columns of the matrix <i>A</i> and of the number of rows of the matrix <i>C</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix <i>B</i> and the number of columns of the matrix <i>C</i> ($n \geq 0$).
<i>a</i>	REAL for clarcm DOUBLE PRECISION for zlarcm Array, DIMENSION(<i>lda</i> , <i>m</i>). Contains the <i>m</i> -by- <i>m</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> , $lda \geq \max(1, m)$.
<i>b</i>	COMPLEX for clarcm DOUBLE COMPLEX for zlarcm Array, DIMENSION(<i>ldb</i> , <i>n</i>). Contains the <i>m</i> -by- <i>n</i> matrix <i>B</i> .
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> , $ldb \geq \max(1, n)$.
<i>ldc</i>	INTEGER. The leading dimension of the output array <i>c</i> , $ldc \geq \max(1, m)$.
<i>rwork</i>	REAL for clarcm DOUBLE PRECISION for zlarcm

Workspace array, DIMENSION $(2*m*n)$.

Output Parameters

c COMPLEX for `clarcm`
 DOUBLE COMPLEX for `zlarcm`
 Array, DIMENSION (ldc, n) . Contains the m -by- n matrix C .

?la_gerpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a general matrix.

Syntax

```
call sla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call dla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call cla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call zla_gerpvgrw( n, ncols, a, lda, af, ldaf )
```

Include Files

- `mkl.fi`

Description

The `?la_gerpvgrw` routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable.

Input Parameters

n INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.

ncols INTEGER. The number of columns of the matrix A ; $ncols \geq 0$.

a, af REAL for `sla_gerpvgrw`
 DOUBLE PRECISION for `dla_gerpvgrw`
 COMPLEX for `cla_gerpvgrw`
 DOUBLE COMPLEX for `zla_gerpvgrw`.
 Arrays: $a(lda, *)$, $af(ldaf, *)$.
 The array a contains the input n -by- n matrix A . The second dimension of a must be at least $\max(1, n)$.
 The array af contains the factors L and U from the factorization triangular factor L or U from the Cholesky factorization $A = P*L*U$ as computed by `?getrf`. The second dimension of af must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*; $ldaf \geq \max(1, n)$.

See Also

[?getrf](#)

[?larscl2](#)

Performs reciprocal diagonal scaling on a vector.

Syntax

call slarscl2(*m*, *n*, *d*, *x*, *ldx*)

call dlarscl2(*m*, *n*, *d*, *x*, *ldx*)

call clarscl2(*m*, *n*, *d*, *x*, *ldx*)

call zlarscl2(*m*, *n*, *d*, *x*, *ldx*)

Include Files

- mkl.fi

Description

The [?larscl2](#) routines perform reciprocal diagonal scaling on a vector

$$x := D^{-1} * x,$$

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m INTEGER. Specifies the number of rows of the matrix *D* and the number of elements of the vector *x*. The value of *m* must be at least zero.

n INTEGER. The number of columns of *D* and *x*. The value of *n* must be at least zero.

d REAL for slarscl2 and clarscl2.
DOUBLE PRECISION for dlarscl2 and zlarscl2.
Array, DIMENSION *m*. Diagonal matrix *D* stored as a vector of length *m*.

x REAL for slarscl2.
DOUBLE PRECISION for dlarscl2.
COMPLEX for clarscl2.
DOUBLE COMPLEX for zlarscl2.
Array, DIMENSION (*ldx*, *n*). The vector *x* to scale by *D*.

ldx INTEGER.

The leading dimension of the vector x . The value of ldx must be at least zero.

Output Parameters

x Scaled vector x .

?lascl2

Performs diagonal scaling on a vector.

Syntax

```
call slascl2(m, n, d, x, ldx)
```

```
call dlascl2(m, n, d, x, ldx)
```

```
call clascl2(m, n, d, x, ldx)
```

```
call zlascl2(m, n, d, x, ldx)
```

Include Files

- mkl.fi

Description

The ?lascl2 routines perform diagonal scaling on a vector

$$x := D * x,$$

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix D and the number of elements of the vector x . The value of m must be at least zero.
n	INTEGER. The number of columns of D and x . The value of n must be at least zero.
d	REAL for slascl2 and clascl2. DOUBLE PRECISION for dlascl2 and zlascl2. Array, DIMENSION m . Diagonal matrix D stored as a vector of length m .
x	REAL for slascl2. DOUBLE PRECISION for dlascl2. COMPLEX for clascl2. DOUBLE COMPLEX for zlascl2. Array, DIMENSION (ldx, n) . The vector x to scale by D .
ldx	INTEGER.

The leading dimension of the vector x . The value of ldx must be at least zero.

Output Parameters

x Scaled vector x .

?la_syamv

Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.

Syntax

```
call sla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call cla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- mkl.fi

Description

The ?la_syamv routines perform a matrix-vector operation defined as

$$y := \alpha \cdot \text{abs}(A) \cdot \text{abs}(x) + \beta \cdot \text{abs}(y),$$

where:

α and β are scalars,

x and y are vectors,

A is an n -by- n Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

$uplo$ CHARACTER*1.
 Specifies whether the upper or lower triangular part of the array A is to be referenced:
 If $uplo = 'BLAS_UPPER'$, only the upper triangular part of A is to be referenced,
 If $uplo = 'BLAS_LOWER'$, only the lower triangular part of A is to be referenced.

n INTEGER. Specifies the number of rows and columns of the matrix A . The value of n must be at least zero.

<i>alpha</i>	<p>REAL for <code>sla_syamv</code> and <code>cla_syamv</code></p> <p>DOUBLE PRECISION for <code>dla_syamv</code> and <code>zla_syamv</code>.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>REAL for <code>sla_syamv</code></p> <p>DOUBLE PRECISION for <code>dla_syamv</code></p> <p>COMPLEX for <code>cla_syamv</code></p> <p>DOUBLE COMPLEX for <code>zla_syamv</code>.</p> <p>Array, DIMENSION(<i>lda</i>, *). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>a</i> must contain the matrix of coefficients. The second dimension of <i>a</i> must be at least $\max(1, n)$.</p>
<i>lda</i>	<p>INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.</p>
<i>x</i>	<p>REAL for <code>sla_syamv</code></p> <p>DOUBLE PRECISION for <code>dla_syamv</code></p> <p>COMPLEX for <code>cla_syamv</code></p> <p>DOUBLE COMPLEX for <code>zla_syamv</code>.</p> <p>Array, DIMENSION at least $(1 + (n-1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the vector <i>X</i>.</p>
<i>incx</i>	<p>INTEGER. Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must be non-zero.</p>
<i>beta</i>	<p>REAL for <code>sla_syamv</code> and <code>cla_syamv</code></p> <p>DOUBLE PRECISION for <code>dla_syamv</code> and <code>zla_syamv</code></p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is zero, you do not need to set <i>y</i> on input.</p>
<i>y</i>	<p>REAL for <code>sla_syamv</code> and <code>cla_syamv</code></p> <p>DOUBLE PRECISION for <code>dla_syamv</code> and <code>zla_syamv</code></p> <p>Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry with non-zero <i>beta</i>, the incremented array <i>y</i> must contain the vector <i>Y</i>.</p>
<i>incy</i>	<p>INTEGER. Specifies the increment for the elements of <i>y</i>.</p> <p>The value of <i>incy</i> must be non-zero.</p>

Output Parameters

<i>y</i>	Updated vector <i>Y</i> .
----------	---------------------------

?la_syrcond

Estimates the Skeel condition number for a symmetric indefinite matrix.

Syntax

```
call sla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- mkl.fi

Description

The function estimates the Skeel condition number of

$$\text{op}(A) * \text{op2}(C)$$

where

the *cmode* parameter determines *op2* as follows:

<i>cmode</i> Value	op2(C)
1	C
0	I
-1	inv(C)

The Skeel condition number

$$\text{cond}(A) = \text{norminf}(|\text{inv}(A)| |A|)$$

is computed by computing scaling factors *R* such that

$$\text{diag}(R) * A * \text{op2}(C)$$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <i>uplo</i> = 'U', the upper triangle of A is stored, If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.
<i>a, af, c, work</i>	REAL for sla_syrcond DOUBLE PRECISION for dla_syrcond Arrays: <i>ab</i> (<i>lda</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix A. <i>af</i> (<i>ldaf</i> ,*) contains the The block diagonal matrix D and the multipliers used to obtain the factor L or U as computed by ?sytrf.

The second dimension of a and af must be at least $\max(1, n)$.
 c , DIMENSION n . The vector c in the formula $\text{op}(A) * \text{op2}(C)$.
 $work$ is a workspace array of DIMENSION $(3*n)$.

lda INTEGER. The leading dimension of the array ab . $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of af . $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by `?sytrf`.

cmode INTEGER. Determines $\text{op2}(C)$ in the formula $\text{op}(A) * \text{op2}(C)$ as follows:

If $cmode = 1$, $\text{op2}(C) = C$.

If $cmode = 0$, $\text{op2}(C) = I$.

If $cmode = -1$, $\text{op2}(C) = \text{inv}(C)$.

iwork INTEGER. Workspace array with DIMENSION n .

Output Parameters

info INTEGER.

If $info = 0$, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

[?sytrf](#)

[?la_syrcond_c](#)

*Computes the infinity norm condition number of $\text{op}(A) * \text{inv}(\text{diag}(c))$ for symmetric indefinite matrices.*

Syntax

```
call cla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

```
call zla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{inv}(\text{diag}(c))$

where the c is a REAL vector for `cla_syrcond_c` and a DOUBLE PRECISION vector for `zla_syrcond_c`.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If $uplo = 'U'$, the upper triangle of A is stored,

If $uplo = 'L'$, the lower triangle of A is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.

a COMPLEX for `cla_syrcond_c`
 DOUBLE COMPLEX for `zla_syrcond_c`
 Array, DIMENSION(lda , *). On entry, the n -by- n matrix A . The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

af COMPLEX for `cla_syrcond_c`
 DOUBLE COMPLEX for `zla_syrcond_c`
 Array, DIMENSION($ldaf$, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by `?sytrf`. The second dimension of af must be at least $\max(1, n)$.

$ldaf$ INTEGER. The leading dimension of the array af . $ldaf \geq \max(1, n)$.

$ipiv$ INTEGER.
 Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by `?sytrf`.

c REAL for `cla_syrcond_c`
 DOUBLE PRECISION for `zla_syrcond_c`
 Array c with DIMENSION n . The vector c in the formula

$$\text{op}(A) * \text{inv}(\text{diag}(c)).$$

$caply$ LOGICAL. If `.TRUE.`, then the function uses the vector c from the formula

$$\text{op}(A) * \text{inv}(\text{diag}(c)).$$

$work$ COMPLEX for `cla_syrcond_c`
 DOUBLE COMPLEX for `zla_syrcond_c`
 Array DIMENSION $2*n$. Workspace.

$rwork$ REAL for `cla_syrcond_c`
 DOUBLE PRECISION for `zla_syrcond_c`
 Array DIMENSION n . Workspace.

Output Parameters

$info$ INTEGER.
 If $info = 0$, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

?sytrf

?la_syrcond_x

Computes the infinity norm condition number of $op(A)*diag(x)$ for symmetric indefinite matrices.

Syntax

```
call cla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

```
call zla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- mkl.fi

Description

The function computes the infinity norm condition number of

$op(A) * diag(x)$

where the x is a COMPLEX vector for `cla_syrcond_x` and a DOUBLE COMPLEX vector for `zla_syrcond_x`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <code>uplo = 'U'</code> , the upper triangle of A is stored, If <code>uplo = 'L'</code> , the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.
<code>a</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array, DIMENSION(<code>lda</code> , *). On entry, the n -by- n matrix A. The second dimension of <code>a</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $lda \geq \max(1, n)$.
<code>af</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array, DIMENSION(<code>ldaf</code> , *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> . $ldaf \geq \max(1, n)$.
<code>ipiv</code>	INTEGER.

Array with `DIMENSIONn`. Details of the interchanges and the block structure of `D` as determined by `?sytrf`.

`x` `COMPLEX` for `cla_syrcond_c`
 `DOUBLE COMPLEX` for `zla_syrcond_c`
 Array `x` with `DIMENSIONn`. The vector `x` in the formula

$$\text{op}(A) * \text{inv}(\text{diag}(x)).$$

`work` `COMPLEX` for `cla_syrcond_c`
 `DOUBLE COMPLEX` for `zla_syrcond_c`
 Array `DIMENSION 2*n`. Workspace.

`rwork` `REAL` for `cla_syrcond_c`
 `DOUBLE PRECISION` for `zla_syrcond_c`
 Array `DIMENSIONn`. Workspace.

Output Parameters

`info` `INTEGER`.
 If `info = 0`, the execution is successful.
 If `i > 0`, the `i`-th parameter is invalid.

See Also

[?sytrf](#)

[?la_syrfsx_extended](#)

Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

```
call sla_syrfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call dla_syrfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call cla_syrfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

```
call zla_syrfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- `mkl.fi`

Description

The `?la_syrfSX_extended` subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?syrfSX` routine calls `?la_syrfSX_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use `?la_syrfSX_extended` to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

<code>prec_type</code>	<p>INTEGER.</p> <p>Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code>, where <code>p</code> is a CHARACTER and:</p> <p>If <code>p = 'S'</code>: Single.</p> <p>If <code>p = 'D'</code>: Double.</p> <p>If <code>p = 'I'</code>: Indigenous.</p> <p>If <code>p = 'X', 'E'</code>: Extra.</p>
<code>uplo</code>	<p>CHARACTER*1. Must be 'U' or 'L'.</p> <p>Specifies the triangle of A to store:</p> <p>If <code>uplo = 'U'</code>, the upper triangle of A is stored,</p> <p>If <code>uplo = 'L'</code>, the lower triangle of A is stored.</p>
<code>n</code>	<p>INTEGER. The number of linear equations; the order of the matrix A; $n \geq 0$.</p>
<code>nrhs</code>	<p>INTEGER. The number of right-hand sides; the number of columns of the matrix B.</p>
<code>a, af, b, y</code>	<p>REAL for <code>sla_syrfSX_extended</code></p> <p>DOUBLE PRECISION for <code>dla_syrfSX_extended</code></p> <p>COMPLEX for <code>cla_syrfSX_extended</code></p> <p>DOUBLE COMPLEX for <code>zla_syrfSX_extended</code>.</p> <p>Arrays: <code>a(lda,*)</code>, <code>af(ldaf,*)</code>, <code>b(ldb,*)</code>, <code>y(ldy,*)</code>.</p> <p>The array <code>a</code> contains the original n-by-n matrix A. The second dimension of <code>a</code> must be at least <code>max(1, n)</code>.</p> <p>The array <code>af</code> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?sytrf</code>.</p> <p>The second dimension of <code>af</code> must be at least <code>max(1, n)</code>.</p> <p>The array <code>b</code> contains the right-hand-side of the matrix B. The second dimension of <code>b</code> must be at least <code>max(1, nrhs)</code>.</p>

The array *y* on entry contains the solution matrix *X* as computed by ?sytrs. The second dimension of *y* must be at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of the array *a*; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of the array *af*; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array with DIMENSION *n*. Details of the interchanges and the block structure of *D* as determined by ?sytrf.

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c REAL for sla_syrfssx_extended and cla_syrfssx_extended
DOUBLE PRECISION for dla_syrfssx_extended and
zla_syrfssx_extended.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; $ldb \geq \max(1, n)$.

ldy INTEGER. The leading dimension of the array *y*; $ldy \geq \max(1, n)$.

n_norms INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If $n_norms \geq 1$, returns normwise error bounds.

If $n_norms \geq 2$, returns componentwise error bounds.

err_bnds_norm REAL for sla_syrfssx_extended and cla_syrfssx_extended
DOUBLE PRECISION for dla_syrfssx_extended and
zla_syrfssx_extended.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_syrfSX_extended/cla_syrfSX_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_syrfSX_extended/zla_syrfSX_extended</code> .
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_syrfSX_extended/cla_syrfSX_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_syrfSX_extended/zla_syrfSX_extended</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_syrfSX_extended/cla_syrfSX_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_syrfSX_extended/zla_syrfSX_extended</code> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.

`err_bnds_comp`

REAL for `sla_syrfSX_extended` and `cla_syrfSX_extended`
DOUBLE PRECISION for `dla_syrfSX_extended` and `zla_syrfSX_extended`.

Array, DIMENSION(*nrhs*, *n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params*(3) = 0.0), then *err_bnds_comp* is not accessed. If *n_err_bnds* < 3, then at most the first (*:,n_err_bnds*) entries are returned.

The first index in *err_bnds_comp*(*i*, :) corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp*(:, *err*) contains the following three fields:

- err*=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt*(*n*)**slamch*(ϵ) for *sla_syrfSX_extended*/*cla_syrfSX_extended* and *sqrt*(*n*)**dLamch*(ϵ) for *dla_syrfSX_extended*/*zla_syrfSX_extended*.
- err*=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt*(*n*)**slamch*(ϵ) for *sla_syrfSX_extended*/*cla_syrfSX_extended* and *sqrt*(*n*)**dLamch*(ϵ) for *dla_syrfSX_extended*/*zla_syrfSX_extended*. This error bound should only be trusted if the previous boolean is true.
- err*=3 Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold *sqrt*(*n*)**slamch*(ϵ) for *sla_syrfSX_extended*/*cla_syrfSX_extended* and *sqrt*(*n*)**dLamch*(ϵ) for *dla_syrfSX_extended*/*zla_syrfSX_extended* to determine if the error estimate is "guaranteed". These reciprocal

condition numbers are $1/(\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

res, dy, y_tail

REAL for `sla_syrfsx_extended`
 DOUBLE PRECISION for `dla_syrfsx_extended`
 COMPLEX for `cla_syrfsx_extended`
 DOUBLE COMPLEX for `zla_syrfsx_extended`.

Workspace arrays of DIMENSION n .

res holds the intermediate residual.

dy holds the intermediate solution.

y_tail holds the trailing bits of the intermediate solution.

ayb

REAL for `sla_syrfsx_extended` and `cla_syrfsx_extended`
 DOUBLE PRECISION for `dla_syrfsx_extended` and
`zla_syrfsx_extended`.

Workspace array, DIMENSION n .

rcond

REAL for `sla_syrfsx_extended` and `cla_syrfsx_extended`
 DOUBLE PRECISION for `dla_syrfsx_extended` and
`zla_syrfsx_extended`.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in *err_bnds_norm* and *err_bnds_comp* may no longer be trustworthy.

rthresh

REAL for `sla_syrfsx_extended` and `cla_syrfsx_extended`
 DOUBLE PRECISION for `dla_syrfsx_extended` and
`zla_syrfsx_extended`.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(dx_{\{i+1\}}) < rthresh * \text{norm}(dx_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$0 < rthresh \leq 1$.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for *sla_syrfssx_extended* and *cla_syrfssx_extended*

DOUBLE PRECISION for *dla_syrfssx_extended* and *zla_syrfssx_extended*.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If `.TRUE.`, the function ignores componentwise convergence. Default value is `.FALSE.`

Output Parameters

y

REAL for *sla_syrfssx_extended*

DOUBLE PRECISION for *dla_syrfssx_extended*

COMPLEX for *cla_syrfssx_extended*

DOUBLE COMPLEX for *zla_syrfssx_extended*.

The improved solution matrix Y .

berr_out

REAL for *sla_syrfssx_extended* and *cla_syrfssx_extended*

DOUBLE PRECISION for *dla_syrfssx_extended* and *zla_syrfssx_extended*.

Array, DIMENSION *nrhs*. *berr_out(j)* contains the componentwise relative backward error for right-hand-side j from the formula

$$\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B)) (i))$$

where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z . This is computed by `?la_lin_berr`.

err_bnds_norm,
err_bnds_comp

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array `(1:nrhs, 2)`. The other elements are kept unchanged.

info

INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = $-i$, the i -th parameter had an illegal value.

See Also

[?syrfssx](#)

[?sytrf](#)

?sytrs
 ?lamch
 ilaprec
 ilatrans
 ?la_lin_berr

?la_syrpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a symmetric indefinite matrix.

Syntax

```
call sla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call dla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call cla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call zla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- mkl.fi

Description

The ?la_syrpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of <i>A</i> to store: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored, If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>info</i>	INTEGER. The value of INFO returned from ?sytrf, that is, the pivot in column <i>info</i> is exactly 0.
<i>a, af</i>	REAL for sla_syrpvgrw DOUBLE PRECISION for dla_syrpvgrw COMPLEX for cla_syrpvgrw DOUBLE COMPLEX for zla_syrpvgrw. Arrays: <i>a</i> (<i>lda</i> ,*), <i>af</i> (<i>ldaf</i> ,*). The array <i>a</i> contains the input <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$.

The array *af* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by `?sytrf`.

The second dimension of *af* must be at least $\max(1, n)$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION <i>n</i> . Details of the interchanges and the block structure of <i>D</i> as determined by <code>?sytrf</code> .
<i>work</i>	REAL for <code>sla_syrpvgrw</code> and <code>cla_syrpvgrw</code> DOUBLE PRECISION for <code>dla_syrpvgrw</code> and <code>zla_syrpvgrw</code> . Workspace array, dimension $2*n$.

See Also

[?sytrf](#)

[?la_wwaddw](#)

Adds a vector into a doubled-single vector.

Syntax

```
call sla_wwaddw( n, x, y, w )
call dla_wwaddw( n, x, y, w )
call cla_wwaddw( n, x, y, w )
call zla_wwaddw( n, x, y, w )
```

Include Files

- `mkl.fi`

Description

The `?la_wwaddw` routine adds a vector *W* into a doubled-single vector (*X*, *Y*). This works for all existing IBM hex and binary floating-point arithmetics, but not for decimal.

Input Parameters

<i>n</i>	INTEGER. The length of vectors <i>X</i> , <i>Y</i> , and <i>W</i> .
<i>x</i> , <i>y</i> , <i>w</i>	REAL for <code>sla_wwaddw</code> DOUBLE PRECISION for <code>dla_wwaddw</code> COMPLEX for <code>cla_wwaddw</code> DOUBLE COMPLEX for <code>zla_wwaddw</code> . Arrays DIMENSION <i>n</i> . <i>x</i> and <i>y</i> contain the first and second parts of the doubled-single accumulation vector, respectively.

w contains the vector W to be added.

Output Parameters

x, y Contain the first and second parts of the doubled-single accumulation vector, respectively, after adding the vector W .

mkl_?tppack

Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.

Syntax

```
call mkl_stppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_dtpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ctppack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ztpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_tppack (ap, i, j, rows, cols, a[, uplo] [, trans] [, info])
```

Include Files

- mkl.fi, lapack.f90

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard full format to packed format

$$AP_{i:i+rows-1, j:j+cols-1} := op(A)$$

Standard packed formats include:

- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE

Any elements of the copied submatrix rectangular outside of the triangular part of the matrix AP are skipped.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether the matrix <i>AP</i> is upper or lower triangular.</p> <p>If <i>uplo</i> = 'U', <i>AP</i> is upper triangular.</p> <p>If <i>uplo</i> = 'L': <i>AP</i> is lower triangular.</p>
<i>trans</i>	<p>CHARACTER*1. Specifies whether or not the copied block of <i>A</i> is transposed or not.</p> <p>If <i>trans</i> = 'N', no transpose: $\text{op}(A) = A$.</p> <p>If <i>trans</i> = 'T', transpose: $\text{op}(A) = A^T$.</p> <p>If <i>trans</i> = 'C', conjugate transpose: $\text{op}(A) = A^H$. For real data this is the same as <i>trans</i> = 'T'.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>AP</i>; $n \geq 0$</p>
<i>i, j</i>	<p>INTEGER. Coordinates of the left upper corner of the destination submatrix in <i>AP</i>.</p> <p>If <i>uplo</i>='U', $1 \leq i \leq j \leq n$.</p> <p>If <i>uplo</i>='L', $1 \leq j \leq i \leq n$.</p>
<i>rows</i>	<p>INTEGER. Number of rows in the destination submatrix. $0 \leq \text{rows} \leq n - i + 1$.</p>
<i>cols</i>	<p>INTEGER. Number of columns in the destination submatrix. $0 \leq \text{cols} \leq n - j + 1$.</p>
<i>a</i>	<p>REAL for <code>mkl_stppack</code></p> <p>DOUBLE PRECISION for <code>mkl_dtpack</code></p> <p>COMPLEX for <code>mkl_ctppack</code></p> <p>DOUBLE COMPLEX for <code>mkl_ztpack</code></p> <p>Pointer to the source submatrix.</p> <p>Array <i>a</i>(<i>lda</i>, *) contains the <i>rows</i>-by-<i>cols</i> submatrix stored as unpacked rows-by-columns if <i>trans</i> = 'N', or unpacked columns-by-rows if <i>trans</i> = 'T' or <i>trans</i> = 'C'. The size of <i>a</i> must be at least <i>lda</i>*<i>cols</i> for <i>trans</i> = 'N' or <i>lda</i>*<i>rows</i> for <i>trans</i>='T' or <i>trans</i>='C'.</p>

NOTE

If there are elements outside of the triangular part of *AP*, they are skipped and are not copied from *a*.

Output Parameters

<i>ap</i>	<p>REAL for <code>mkl_stppack</code></p> <p>DOUBLE PRECISION for <code>mkl_dtpack</code></p>
-----------	--

COMPLEX for `mkl_ctppack`

DOUBLE COMPLEX for `mkl_ztppack`

Array of size at least $\max(1, n(n+1)/2)$. The array `ap` contains either the upper or the lower triangular part of the matrix AP (as specified by `uplo`) in packed storage (see [Matrix Storage Schemes](#)). The submatrix of `ap` from row i to row $i + rows - 1$ and column j to column $j + cols - 1$ is overwritten with a copy of the source matrix.

`info`

INTEGER. If `info=0`, the execution is successful. If `info = -i`, the i -th parameter had an illegal value.

mkl_?tpunpack

Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.

Syntax

```
call mkl_stpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_dtpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ctpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_ztpunpack (uplo, trans, n, ap, i, j, rows, cols, a, lda, info )
call mkl_tpunpack (ap, i, j, rows, cols, a[, uplo] [, trans] [, info])
```

Include Files

- `mkl.fi`, `lapack.f90`

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard packed format to full format.

$A := \text{op}(AP_{i:i+rows-1, j:j+cols-1})$

Standard packed formats include:

- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE

Any elements of the copied submatrix rectangular outside of the triangular part of AP are skipped.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	<p>CHARACTER*1. Specifies whether matrix <i>AP</i> is upper or lower triangular.</p> <p>If <i>uplo</i> = 'U', <i>AP</i> is upper triangular.</p> <p>If <i>uplo</i> = 'L': <i>AP</i> is lower triangular.</p>
<i>trans</i>	<p>CHARACTER*1. Specifies whether or not the copied block of <i>AP</i> is transposed.</p> <p>If <i>trans</i> = 'N', no transpose: $op(AP) = AP$.</p> <p>If <i>trans</i> = 'T', transpose: $op(AP) = AP^T$.</p> <p>If <i>trans</i> = 'C', conjugate transpose: $op(AP) = AP^H$. For real data this is the same as <i>trans</i> = 'T'.</p>
<i>n</i>	<p>INTEGER. The order of the matrix <i>AP</i>; $n \geq 0$.</p>
<i>ap</i>	<p>REAL for <code>mkl_stpunpack</code></p> <p>DOUBLE PRECISION for <code>mkl_dtpunpack</code></p> <p>COMPLEX for <code>mkl_ctpunpack</code></p> <p>DOUBLE COMPLEX for <code>mkl_ztpunpack</code></p> <p>Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains either the upper or the lower triangular part of the matrix <i>AP</i> (as specified by <i>uplo</i>) in packed storage (see Matrix Storage Schemes). It is the source for the submatrix of <i>AP</i> from row <i>i</i> to row <i>i</i> + <i>rows</i> - 1 and column <i>j</i> to column <i>j</i> + <i>cols</i> - 1 to be copied.</p>
<i>i, j</i>	<p>INTEGER. Coordinates of left upper corner of the submatrix in <i>AP</i> to copy.</p> <p>If <i>uplo</i>='U', $1 \leq i \leq j \leq n$.</p> <p>If <i>uplo</i>='L', $1 \leq j \leq i \leq n$.</p>
<i>rows</i>	<p>INTEGER. Number of rows to copy. $0 \leq rows \leq n - i + 1$.</p>
<i>cols</i>	<p>INTEGER. Number of columns to copy. $0 \leq cols \leq n - j + 1$.</p>
<i>lda</i>	<p>INTEGER. The leading dimension of array <i>a</i>.</p> <p>$lda \geq \max(1, rows)$ for <i>trans</i> = 'N' and $lda \geq \max(1, cols)$ for <i>trans</i>='T' or <i>trans</i>='C'.</p>

Output Parameters

<i>a</i>	<p>REAL for <code>mkl_stpunpack</code></p> <p>DOUBLE PRECISION for <code>mkl_dtpunpack</code></p> <p>COMPLEX for <code>mkl_ctpunpack</code></p> <p>DOUBLE COMPLEX for <code>mkl_ztpunpack</code></p>
----------	--

Pointer to the destination matrix. The size of a must be at least $lda*cols$ for $trans = 'N'$ or $lda*rows$ for $trans='T'$ or $trans='C'$. On exit, array a is overwritten with a copy of the unpacked $rows$ -by- $cols$ submatrix of ap unpacked rows-by-columns if $trans = 'N'$, or unpacked columns-by-rows if $trans = 'T'$ or $trans = 'C'$.

NOTE

If there are elements outside of the triangular part of ap indicated by $uplo$, they are skipped and are not copied to a .

info

INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Utility Functions and Routines

This section describes LAPACK utility functions and routines.

Summary information about these routines is given in the following table:

LAPACK Utility Routines

Routine Name	Data Types	Description
<code>ilaver</code>		Returns the version of the Lapack library.
<code>ilaenv</code>		Environmental enquiry function which returns values for tuning algorithmic performance.
<code>iparmq</code>		Environmental enquiry function which returns values for tuning algorithmic performance.
<code>ieeeck</code>		Checks if the infinity and NaN arithmetic is safe. Called by <code>ilaenv</code> .
<code>?labad</code>	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
<code>?lamch</code>	s, d	Determines machine parameters for floating-point arithmetic.
<code>?lamc1</code>	s, d	Called from <code>?lamc2</code> . Determines machine parameters given by β , t , rnd , $ieee1$.
<code>?lamc2</code>	s, d	Used by <code>?lamch</code> . Determines machine parameters specified in its arguments list.
<code>?lamc3</code>	s, d	Called from <code>?lamc1</code> - <code>?lamc5</code> . Intended to force a and b to be stored prior to doing the addition of a and b .
<code>?lamc4</code>	s, d	This is a service routine for <code>?lamc2</code> .
<code>?lamc5</code>	s, d	Called from <code>?lamc2</code> . Attempts to compute the largest machine floating-point number, without overflow.
<code>chla_transtype</code>		Translates a BLAST-specified integer constant to the character string specifying a transposition operation.
<code>iladiag</code>		Translates a character string specifying whether a matrix has a unit diagonal or not to the relevant BLAST-specified integer constant.

Routine Name	Data Types	Description
<code>ilaprec</code>		Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.
<code>ilatrans</code>		Translates a character string specifying a transposition operation to the BLAST-specified integer constant.
<code>ilauplo</code>		Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.
<code>xerbla_array</code>		Assists other languages in calling the <code>xerbla</code> function.

See Also

`lsame` Tests two characters for equality regardless of the case.

`lsamen` Tests two character strings for equality regardless of the case.

`second/dsecond` Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

`xerbla` Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

ilaver

Returns the version of the LAPACK library.

Syntax

```
call ilaver( vers_major, vers_minor, vers_patch )
```

Include Files

- `mkl.fi`

Description

This routine returns the version of the LAPACK library.

Output Parameters

<code>vers_major</code>	INTEGER. Returns the major version of the LAPACK library.
<code>vers_minor</code>	INTEGER. Returns the minor version from the major version of the LAPACK library.
<code>vers_patch</code>	INTEGER. Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function that returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv( ispec, name, opts, n1, n2, n3, n4 )
```

Include Files

- `mkl.fi`

Description

The enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* below for a description of the parameters.

This version provides a set of parameters that should give good, but not optimal, performance on many of the currently available computers.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

ispec

INTEGER.

Specifies the parameter to be returned as the value of `ilaenv`:

= 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.

= 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.

= 3: the crossover point (in a block routine, for n less than this value, an unblocked routine should be used)

= 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)

= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k -by- m , where k is given by `ilaenv(2, ...)` and m by `ilaenv(5, ...)`

= 6: the crossover point for the SVD (when reducing an m -by- n matrix to bidiagonal form, if $\max(m, n) / \min(m, n)$ exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form.)

= 7: the number of processors

= 8: the crossover point for the multishift QR and QZ methods for nonsymmetric eigenvalue problems (deprecated).

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?ge1sd` and `?gesdd`)

=10: ieee NaN arithmetic can be trusted not to trap

=11: infinity arithmetic can be trusted not to trap

12 $\leq ispec \leq 16$: ?hseqr or one of its subroutines, see iparmq for detailed explanation.

name CHARACTER*(*). The name of the calling subroutine, in either upper case or lower case.

opts CHARACTER*(*). The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.

n1, n2, n3, n4 INTEGER. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value INTEGER.
 If *value* ≥ 0 : the value of the parameter specified by *ispec*;
 If *value* = -k < 0: the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling *ilaenv* from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1, n2, n3, n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by *ilaenv* is checked for validity in the calling subroutine. For example, *ilaenv* is used to retrieve the optimal blocksize for *strtri* as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1 )
```

```
if( nb.le.1 ) nb = max( 1, n )
```

See Also

?hseqr
 iparmq

iparmq

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

```
value = iparmq( ispec, name, opts, n, ilo, ihi, lwork )
```

Include Files

- mkl.fi

Description

The function sets problem and machine dependent parameters useful for ?hseqr and its subroutines. It is called whenever `ilaenv` is called with $12 \leq ispec \leq 16$.

Input Parameters

<i>ispec</i>	INTEGER. Specifies the parameter to be returned as the value of <code>iparmq</code> : = 12: (<i>inmin</i>) Matrices of order <i>nmin</i> or less are sent directly to ?lahqr, the implicit double shift QR algorithm. <i>nmin</i> must be at least 11. = 13: (<i>inwin</i>) Size of the deflation window. This is best set greater than or equal to the number of simultaneous shifts <i>ns</i> . Larger matrices benefit from larger deflation windows. = 14: (<i>inibl</i>) Determines when to stop nibbling and invest in an (expensive) multi-shift QR sweep. If the aggressive early deflation subroutine finds <i>ld</i> converged eigenvalues from an order <i>nw</i> deflation window and $ld > (nw * nibble) / 100$, then the next QR sweep is skipped and early deflation is applied immediately to the remaining active diagonal block. Setting <code>iparmq(ispec=14)=0</code> causes TTQRE to skip a multi-shift QR sweep whenever early deflation finds a converged eigenvalue. Setting <code>iparmq(ispec=14)</code> greater than or equal to 100 prevents TTQRE from skipping a multi-shift QR sweep. = 15: (<i>nshfts</i>) The number of simultaneous shifts in a multi-shift QR iteration. = 16: (<i>iacc22</i>) <code>iparmq</code> is set to 0, 1 or 2 with the following meanings. 0: During the multi-shift QR sweep, ?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update the far-from-diagonal matrix entries. 1: During the multi-shift QR sweep, ?laqr5 and/or ?laqr3 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. 2: During the multi-shift QR sweep, ?laqr5 accumulates reflections and takes advantage of 2-by-2 block structure during matrix-matrix multiplies. (If ?trrm is slower than ?gemm, then <code>iparmq(ispec=16)=1</code> may be more efficient than <code>iparmq(ispec=16)=2</code> despite the greater level of arithmetic work implied by the latter choice.)
<i>name</i>	CHARACTER* (*). The name of the calling subroutine.
<i>opts</i>	CHARACTER* (*). This is a concatenation of the string arguments to TTQRE.
<i>n</i>	INTEGER. <i>n</i> is the order of the Hessenberg matrix <i>H</i> .
<i>ilo, ihi</i>	INTEGER. It is assumed that <i>H</i> is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$.
<i>lwork</i>	INTEGER.

The amount of workspace available.

Output Parameters

value INTEGER.
 If $value \geq 0$: the value of the parameter specified by *iparmq*;
 If $value = -k < 0$: the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1 )
```

```
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe.
 Called by `ilaenv`.

Syntax

```
ival = ieeck( ispec, zero, one )
```

Include Files

- `mkl.fi`

Description

The function `ieeeck` is called from `ilaenv` to verify that infinity and possibly NaN arithmetic is safe, that is, will not trap.

Input Parameters

ispec INTEGER.
 Specifies whether to test just for infinity arithmetic or both for infinity and NaN arithmetic:
 If *ispec* = 0: Verify infinity arithmetic only.
 If *ispec* = 1: Verify infinity and NaN arithmetic.

zero REAL. Must contain the value 0.0
 This is passed to prevent the compiler from optimizing away this code.

one REAL. Must contain the value 1.0
 This is passed to prevent the compiler from optimizing away this code.

Output Parameters

ival INTEGER.
 If *ival* = 0: Arithmetic failed to produce the correct answers.
 If *ival* = 1: Arithmetic produced the correct answers.

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

call slabad(*small*, *large*)

call dlabad(*small*, *large*)

Include Files

- mkl.fi

Description

The routine takes as input the values computed by `slamch/dlamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `?lamch`. This subroutine is needed because `?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

small REAL for slabad
 DOUBLE PRECISION for dlabad.
 The underflow threshold as computed by `?lamch`.

large REAL for slabad
 DOUBLE PRECISION for dlabad.
 The overflow threshold as computed by `?lamch`.

Output Parameters

small On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of *small*, otherwise unchanged.

large On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of *large*, otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch( cmach )
```

```
val = dlamch( cmach )
```

Include Files

- mkl.fi

Description

The function ?lamch determines single precision and double precision machine parameters.

Input Parameters

cmach CHARACTER*1. Specifies the value to be returned by ?lamch:

- = 'E' or 'e', *val* = *eps*
- = 'S' or 's', *val* = *sfmin*
- = 'B' or 'b', *val* = *base*
- = 'P' or 'p', *val* = *eps*base*
- = 'n' or 'n', *val* = *t*
- = 'R' or 'r', *val* = *rnd*
- = 'M' or 'm', *val* = *emin*
- = 'U' or 'u', *val* = *rmin*
- = 'L' or 'l', *val* = *emax*
- = 'O' or 'o', *val* = *rmax*

where

eps = relative machine precision;

sfmin = safe minimum, such that $1/sfmin$ does not overflow;

base = base of the machine;

prec = $eps*base$;

t = number of (base) digits in the mantissa;

rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;

emin = minimum exponent before (gradual) underflow;

rmin = $underflow_threshold - base**(emin-1)$;

emax = largest exponent before overflow;

rmax = $overflow_threshold - (base**emax)*(1-eps)$.

NOTE

You can use a character string for *cmach* instead of a single character in order to make your code more readable. The first character of the string determines the value to be returned. For example, 'Precision' is interpreted as 'p'.

Output Parameters

val REAL for *slamch*
 DOUBLE PRECISION for *dlamch*
 Value returned by the function.

?lamc1

Called from ?lamc2. Determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.

Syntax

```
call slamc1( beta, t, rnd, ieee1 )
call dlamc1( beta, t, rnd, ieee1 )
```

Include Files

- mkl.fi

Description

The routine ?lamc1 determines machine parameters given by *beta*, *t*, *rnd*, *ieee1*.

Output Parameters

beta INTEGER. The base of the machine.

t INTEGER. The number of (*beta*) digits in the mantissa.

rnd LOGICAL.
 Specifies whether proper rounding (*rnd* = .TRUE.) or chopping (*rnd* = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.

ieee1 LOGICAL.
 Specifies whether rounding appears to be done in the *ieee* 'round to nearest' style.

?lamc2

Used by ?lamch. Determines machine parameters specified in its arguments list.

Syntax

```
call slamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

```
call dlamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Include Files

- mkl.fi

Description

The routine ?lamc2 determines machine parameters specified in its arguments list.

Output Parameters

<i>beta</i>	INTEGER. The base of the machine.
<i>t</i>	INTEGER. The number of (<i>beta</i>) digits in the mantissa.
<i>rnd</i>	LOGICAL. Specifies whether proper rounding (<i>rnd</i> = .TRUE.) or chopping (<i>rnd</i> = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
<i>eps</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest positive number such that $fl(1.0 - eps) < 1.0$, where <i>fl</i> denotes the computed value.
<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow occurs.
<i>rmin</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest normalized number for the machine, given by $base^{emin-1}$, where <i>base</i> is the floating point value of <i>beta</i> .
<i>emax</i>	INTEGER. The maximum exponent before overflow occurs.
<i>rmax</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The largest positive number for the machine, given by $base^{emax(1 - eps)}$, where <i>base</i> is the floating point value of <i>beta</i> .

?lamc3

Called from ?lamc1-?lamc5. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.

Syntax

```
val = slamc3( a, b )
```

```
val = dlamc3( a, b )
```

Include Files

- mkl.fi

Description

The routine is intended to force A and B to be stored prior to doing the addition of A and B , for use in situations where optimizers might hold one of these in a register.

Input Parameters

a, b REAL for slamc3
DOUBLE PRECISION for dlamc3
The values a and b .

Output Parameters

val REAL for slamc3
DOUBLE PRECISION for dlamc3
The result of adding values a and b .

?lamc4

This is a service routine for ?lamc2.

Syntax

```
call slamc4( emin, start, base )
```

```
call dlamc4( emin, start, base )
```

Include Files

- mkl.fi

Description

This is a service routine for ?lamc2.

Input Parameters

$start$ REAL for slamc4
DOUBLE PRECISION for dlamc4
The starting point for determining $emin$.

$base$ INTEGER. The base of the machine.

Output Parameters

emin INTEGER. The minimum exponent before (gradual) underflow, computed by setting $a = start$ and dividing by $base$ until the previous a can not be recovered.

?lamc5

Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5( beta, p, emin, ieee, emax, rmax)
```

```
call dlamc5( beta, p, emin, ieee, emax, rmax)
```

Include Files

- mkl.fi

Description

The routine ?lamc5 attempts to compute $rmax$, the largest machine floating-point number, without overflow. It assumes that $emax + abs(emin)$ sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 ($emin = -28625$, $emax = 28718$). It will also fail if the value supplied for $emin$ is too large (that is, too close to zero), probably with overflow.

Input Parameters

beta INTEGER. The base of floating-point arithmetic.

p INTEGER. The number of base $beta$ digits in the mantissa of a floating-point value.

emin INTEGER. The minimum exponent before (gradual) underflow.

ieee LOGICAL. A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters

emax INTEGER. The largest exponent before overflow.

rmax REAL for slamc5
DOUBLE PRECISION for dlamc5
The largest machine floating-point number.

chla_transtype

Translates a BLAST-specified integer constant to the character string specifying a transposition operation.

Syntax

```
val = chla_transtype( trans )
```


Output Parameters

val INTEGER
Value returned by the function.

ilaprec

Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

Syntax

```
val = ilaprec( prec )
```

Include Files

- mkl.fi

Description

The `ilaprec` function translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

The function returns an `INTEGER`. If `val < 0`, the input is not a character indicating a supported intermediate precision. Otherwise, the function returns the constant value corresponding to `prec`.

Input Parameters

prec CHARACTER*1.
Specifies the form of the system of equations:
If `prec = 'S'`: Single.
If `prec = 'D'`: Double.
If `prec = 'I'`: Indigenous.
If `prec = 'X', 'E'`: Extra.

Output Parameters

val INTEGER
Value returned by the function.

ilatrans

Translates a character string specifying a transposition operation to the BLAST-specified integer constant.

Syntax

```
val = ilatrans( trans )
```

Include Files

- mkl.fi

Description

The `ilatrans` function translates a character string specifying a transposition operation to the BLAST-specified integer constant.

The function returns a `INTEGER`. If `val < 0`, the input is not a character indicating a transposition operator. Otherwise, the function returns the constant value corresponding to `trans`.

Input Parameters

`trans` CHARACTER*1.
 Specifies the form of the system of equations:
 If `trans = 'N'`: No transpose.
 If `trans = 'T'`: Transpose.
 If `trans = 'C'`: Conjugate Transpose.

Output Parameters

`val` INTEGER
 Character that specifies a transposition operation.

ilauplo

Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

Syntax

```
val = ilauplo( uplo )
```

Include Files

- `mkl.fi`

Description

The `ilauplo` function translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

The function returns an `INTEGER`. If `val < 0`, the input is not a character indicating an upper- or lower-triangular matrix. Otherwise, the function returns the constant value corresponding to `uplo`.

Input Parameters

`diag` CHARACTER.
 Specifies the form of the system of equations:
 If `diag = 'U'`: *A* is upper triangular.
 If `diag = 'L'`: *A* is lower triangular.

Output Parameters

`val` INTEGER

Value returned by the function.

xerbla_array

Assists other languages in calling the xerbla function.

Syntax

```
call xerbla_array( sname_array, sname_len, info )
```

Include Files

- mkl.fi

Description

The routine assists other languages in calling the error handling `xerbla` function. Rather than taking a Fortran string argument as the function name, `xerbla_array` takes an array of single characters along with the array length. The routine then copies up to 32 characters of that array into a Fortran string and passes that to `xerbla`. If called with a non-positive `sname_len`, the routine will call `xerbla` with a string of all blank characters.

If some macro or other device makes `xerbla_array` available to C99 by a name `lapack_xerbla` and with a common Fortran calling convention, a C99 program could invoke `xerbla` via:

```
{
  int flen = strlen(__func__);
  lapack_xerbla(__func__, &flen, &info);
}
```

Providing `xerbla_array` is not necessary for intercepting LAPACK errors. `xerbla_array` calls `xerbla`.

Output Parameters

<code>sname_array</code>	CHARACTER(1).
	Array, dimension (<code>sname_len</code>). The name of the routine that called <code>xerbla_array</code> .
<code>sname_len</code>	INTEGER.
	The length of the name in <code>sname_array</code> .
<code>info</code>	INTEGER.
	Position of the invalid parameter in the parameter list of the calling routine.

Test Functions and Routines

This section describes LAPACK test functions and routines.

Summary information about these routines is given in the following table:

LAPACK Test Routines

Routine Name	Data Types	Description
<code>?lagge</code>	s, d, c, z	Generates a general m -by- n matrix .

Routine Name	Data Types	Description
?laghe	c, z	Generates a complex Hermitian matrix .
?lagsy	s, d, c, z	Generates a symmetric matrix by pre- and post- multiplying a real diagonal matrix with a random unitary matrix .
?latms	s, d, c, z	Generates a general m -by- n matrix with specific singular values.

?latms

Generates a general m -by- n matrix with specific singular values.

Syntax

```
call slatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

```
call dlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

```
call clatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

```
call zlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

Description

The ?latms routine generates random matrices with specified singular values, or symmetric/Hermitian matrices with specified eigenvalues for testing LAPACK programs.

It applies this sequence of operations:

1. Set the diagonal to d , where d is input or computed according to $mode$, $cond$, $dmax$, and sym as described in Input Parameters.
2. Generate a matrix with the appropriate band structure, by one of two methods:

Method A

1. Generate a dense m -by- n matrix by multiplying d on the left and the right by random unitary matrices, then:
2. Reduce the bandwidth according to kl and ku , using Householder transformations.

Method B:

Convert the bandwidth-0 (i.e., diagonal) matrix to a bandwidth-1 matrix using Givens rotations, "chasing" out-of-band elements back, much as in QR; then convert the bandwidth-1 to a bandwidth-2 matrix, etc.

Note that for reasonably small bandwidths (relative to m and n) this requires less storage, as a dense matrix is not generated. Also, for symmetric or Hermitian matrices, only one triangle is generated.

Method A is chosen if the bandwidth is a large fraction of the order of the matrix, and lda is at least m (so a dense matrix can be stored.) Method B is chosen if the bandwidth is small (less than $(1/2)*n$ for symmetric or Hermitian or less than $.3*n+m$ for nonsymmetric), or lda is less than m and not less than the bandwidth.

Pack the matrix if desired, using one of the methods specified by the $pack$ parameter.

If Method B is chosen and band format is specified, then the matrix is generated in the band format and no repacking is necessary.

Input Parameters

The data types are given for the Fortran interface.

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix A ($n \geq 0$).
<i>dist</i>	<p>CHARACTER*1. Specifies the type of distribution to be used to generate the random singular values or eigenvalues:</p> <ul style="list-style-type: none"> • 'U': uniform distribution (0, 1) • 'S': symmetric uniform distribution (-1, 1) • 'N': normal distribution (0, 1)
<i>iseed</i>	<p>INTEGER. Array with size 4.</p> <p>Specifies the seed of the random number generator. Values should lie between 0 and 4095 inclusive, and <i>iseed</i>(4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of the array are modified, and can be used in the next call to ?latms to continue the same random number sequence.</p>
<i>sym</i>	<p>CHARACTER*1.</p> <p>If <i>sym</i>='S' or 'H', the generated matrix is symmetric or Hermitian, with eigenvalues specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>; they can be positive, negative, or zero.</p> <p>If <i>sym</i>='P', the generated matrix is symmetric or Hermitian, with eigenvalues (which are singular, non-negative values) specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>.</p> <p>If <i>sym</i>='N', the generated matrix is nonsymmetric, with singular, non-negative values specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>.</p>
<i>d</i>	<p>REAL for slatms and clatms</p> <p>DOUBLE PRECISION for dlatms and zlatms</p> <p>Array, size (MIN(<i>m</i>, <i>n</i>))</p> <p>This array is used to specify the singular values or eigenvalues of A (see the description of <i>sym</i>). If <i>mode</i>=0, then <i>d</i> is assumed to contain the eigenvalues or singular values, otherwise elements of <i>d</i> are computed according to <i>mode</i>, <i>cond</i>, and <i>dmax</i>.</p>
<i>mode</i>	<p>INTEGER. Describes how the singular/eigenvalues are specified.</p> <ul style="list-style-type: none"> • <i>mode</i> = 0: use <i>d</i> as input • <i>mode</i> = 1: set $d(1) = 1$ and $d(2:n) = 1.0/cond$ • <i>mode</i> = 2: set $d(1:n-1) = 1$ and $d(n) = 1.0/cond$ • <i>mode</i> = 3: set $d(i) = cond^{-(i-1)/(n-1)}$ • <i>mode</i> = 4: set $d(i) = 1 - (i-1)/(n-1) * (1 - 1/cond)$

- $mode = 5$: set elements of d to random numbers in the range $(1/cond, 1)$ such that their logarithms are uniformly distributed.
- $mode = 6$: set elements of d to random numbers from same distribution as the rest of the matrix.

$mode < 0$ has the same meaning as $ABS(mode)$, except that the order of the elements of d is reversed. Thus, if $mode$ is positive, d has entries ranging from 1 to $1/cond$, if negative, from $1/cond$ to 1.

If $sym='S'$ or $'H'$, and $mode$ is not 0, 6, nor -6, then the elements of d are also given a random sign (multiplied by +1 or -1).

cond

REAL for slatms and clatms

DOUBLE PRECISION for dlatms and zlatms

Used in setting d as described for the $mode$ parameter. If used, $cond \geq 1$.*dmax*

REAL for slatms and clatms

DOUBLE PRECISION for dlatms and zlatms

If $mode$ is not -6, 0 nor 6, the contents of d , as computed according to $mode$ and $cond$, are scaled by $dmax / \max(abs(d(i)))$; thus, the maximum absolute eigenvalue or singular value (the norm) is $abs(dmax)$.

NOTE

$dmax$ need not be positive: if $dmax$ is negative (or zero), d will be scaled by a negative number (or zero).

kl

INTEGER. Specifies the lower bandwidth of the matrix. For example, $kl=0$ implies upper triangular, $kl=1$ implies upper Hessenberg, and kl being at least $m - 1$ means that the matrix has full lower bandwidth. kl must equal ku if the matrix is symmetric or Hermitian.

ku

INTEGER. Specifies the upper bandwidth of the matrix. For example, $ku=0$ implies lower triangular, $ku=1$ implies lower Hessenberg, and ku being at least $n - 1$ means that the matrix has full upper bandwidth. kl must equal ku if the matrix is symmetric or Hermitian.

pack

CHARACTER*1. Specifies packing of matrix:

- 'N': no packing
- 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
- 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
- 'B': store the lower triangle in band storage scheme (only if matrix symmetric, Hermitian, or lower triangular)
- 'Q': store the upper triangle in band storage scheme (only if matrix symmetric, Hermitian, or upper triangular)
- 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store A in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

	'Z'	'B'	'Q'	'C'	'R'
GB: general band	x				
PB: symmetric positive definite band		x	x		
SB: symmetric band		x	x		
HB: Hermitian band		x	x		
TB: triangular band		x	x		
PP: symmetric positive definite packed				x	x
SP: symmetric packed				x	x
HP: Hermitian packed				x	x
TP: triangular packed				x	x

If two calls to ?latms differ only in the *pack* parameter, they generate mathematically equivalent matrices.

lda

INTEGER. *lda* specifies the first dimension of *a* as declared in the calling program.

If *pack*='N', 'U', 'L', 'C', or 'R', then *lda* must be at least *m*.

If *pack*='B' or 'Q', then *lda* must be at least $\text{MIN}(kl, m - 1)$ (which is equal to $\text{MIN}(ku, n - 1)$).

If *pack*='Z', *lda* must be large enough to hold the packed array: $\text{MIN}(ku, n - 1) + \text{MIN}(kl, m - 1) + 1$.

Output Parameters

iseed

The array *iseed* contains the updated seed.

d

The array *d* contains the updated seed.

NOTE

The array *d* is not modified if *mode* = 0.

a

REAL for slatms

DOUBLE PRECISION for dlatms

COMPLEX for clatms

DOUBLE COMPLEX for zlatms

Array of size *lda* by *n*.

The array *a* contains the generated *m*-by-*n* matrix *A*.

a is first generated in full (unpacked) form, and then packed, if so specified by *pack*. Thus, the first *m* elements of the first *n* columns are always modified. If *pack* specifies a packed or banded storage scheme, all *lda*

elements of the first n columns are modified; the elements of the array which do not correspond to elements of the generated matrix are set to zero.

work

REAL for slatms

DOUBLE PRECISION for dlatms

COMPLEX for clatms

DOUBLE COMPLEX for zlatms

Array of size $(3*\text{MAX}(n, m))$

Workspace.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

If *info* = 2, cannot scale to *dmax* (maximum singular value is 0).

If *info* = 3, error return from [lagge](#), [?laghe](#), or [lagsy](#).

ScaLAPACK Routines

This chapter describes the Intel® Math Kernel Library implementation of routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Intel MKL ScaLAPACK routines are written in FORTRAN 77 with exception of a few utility routines written in C to exploit the IEEE arithmetic. All routines are available in all precision types: single precision, double precision, complex, and double complex precision. See the `mkl_scalapack.h` header file for C declarations of ScaLAPACK routines.

NOTE

ScaLAPACK routines are provided only for Intel® 64 or Intel® Many Integrated Core architectures.

Sections in this chapter include descriptions of ScaLAPACK [computational routines](#) that perform distinct computational tasks, as well as [driver routines](#) for solving standard types of problems in one call.

The `<install_directory>/examples/scalapackf` directory contains sample code demonstrating the use of ScaLAPACK routines.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel MKL version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements, see *Intel® MKL Release Notes* and *Intel® MKL User's Guide*.

For full reference on ScaLAPACK routines and related information, see [\[SLUG\]](#).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Overview

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, and also gives the opportunity to use BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global matrix (array) and its corresponding process and memory location is contained in the array called the *array descriptor* associated with each global matrix. The size of the array descriptor is denoted as *dlen_*.

Let A be a two-dimensional block cyclicly distributed matrix with the array descriptor array $desca$. The meaning of each array descriptor element depends on the type of the matrix A . The tables [Array descriptor for dense matrices](#) and [Array descriptor for narrow-band and tridiagonal matrices](#) describe the meaning of each element for the different types of matrices.

Array descriptor for dense matrices ($dlen_=9$)

Element Name	Stored in	Description	Element Index Number
$dtype_a$	$desca(dtype_)$	Descriptor type (=1 for dense matrices).	1
$ctxt_a$	$desca(ctxt_)$	BLACS context handle for the process grid.	2
m_a	$desca(m_)$	Number of rows in the global matrix A .	3
n_a	$desca(n_)$	Number of columns in the global matrix A .	4
mb_a	$desca(mb_)$	Row blocking factor.	5
nb_a	$desca(nb_)$	Column blocking factor.	6
$rsrc_a$	$desca(rsrc_)$	Process row over which the first row of the global matrix A is distributed.	7
$csrc_a$	$desca(csrc_)$	Process column over which the first column of the global matrix A is distributed.	8
lld_a	$desca(lld_)$	Leading dimension of the local matrix A .	9

Array descriptor for narrow-band and tridiagonal matrices ($dlen_=7$)

Element Name	Stored in	Description	Element Index Number
$dtype_a$	$desca(dtype_)$	Descriptor type <ul style="list-style-type: none"> • $dtype_a=501$: 1-by-P grid, • $dtype_a=502$: P-by-1 grid. 	1
$ctxt_a$	$desca(ctxt_)$	BLACS context handle indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) can vary.	2
n_a	$desca(n_)$	The size of the matrix dimension being distributed.	3
nb_a	$desca(nb_)$	The blocking factor used to distribute the distributed dimension of the matrix A .	4
src_a	$desca(src_)$	The process row or column over which the first row or column of the matrix A is distributed.	5
lld_a	$desca(lld_)$	The leading dimension of the local matrix storing the local blocks of the distributed matrix A . The minimum value of lld_a depends on $dtype_a$. <ul style="list-style-type: none"> • $dtype_a=501$: $lld_a \geq \max(\text{size of undistributed dimension}, 1)$, • $dtype_a=502$: $lld_a \geq \max(nb_a, 1)$. 	6
Not applicable		Reserved for future use.	7

Similar notations are used for different matrices. For example: lld_b is the leading dimension of the local matrix storing the local blocks of the distributed matrix B and $dtype_z$ is the type of the global matrix Z .

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix $sub(A)$ of the global matrix A defined by the following 6 values (for dense matrices):

- m The number of rows of $sub(A)$
- n The number of columns of $sub(A)$

<i>a</i>	A pointer to the local matrix containing the entire global matrix <i>A</i>
<i>ia</i>	The row index of sub(<i>A</i>) in the global matrix <i>A</i>
<i>ja</i>	The column index of sub(<i>A</i>) in the global matrix <i>A</i>
<i>desca</i>	The array descriptor for the global matrix <i>A</i>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Routine Naming Conventions

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines. A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter *p*.

ScaLAPACK names have the structure *p?yyzzz* or *p?yyzz*, which is described below.

The initial letter *p* is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol *?* indicates the data type:

<i>s</i>	real, single precision
<i>d</i>	real, double precision
<i>c</i>	complex, single precision
<i>z</i>	complex, double precision

The second and third letters *yy* indicate the matrix type as:

<i>ge</i>	general
<i>gb</i>	general band
<i>gg</i>	a pair of general matrices (for a generalized problem)
<i>dt</i>	general tridiagonal (diagonally dominant-like)
<i>db</i>	general band (diagonally dominant-like)
<i>po</i>	symmetric or Hermitian positive-definite
<i>pb</i>	symmetric or Hermitian positive-definite band
<i>pt</i>	symmetric or Hermitian positive-definite tridiagonal
<i>sy</i>	symmetric

st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

sv	a <i>simple</i> driver for solving a linear system
svx	an <i>expert</i> driver for solving a linear system
ls	a driver for solving a linear least squares problem
ev	a simple driver for solving a symmetric eigenvalue problem
evd	a simple driver for solving an eigenvalue problem using a divide and conquer algorithm
evx	an expert driver for solving a symmetric eigenvalue problem
svd	a driver for computing a singular value decomposition
gvx	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- [Solving Systems of Linear Equations](#)
- [Orthogonal Factorizations and LLS Problems](#)
- [Symmetric Eigenproblems](#)
- [Nonsymmetric Eigenproblems](#)
- [Singular Value Decomposition](#)
- [Generalized Symmetric-Definite Eigenproblems](#)

See also the respective [driver routines](#).

Linear Equations

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded

- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the LU factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See [Table “Computational Routines for Systems of Linear Equations”](#) for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding [driver routine](#) that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf` (LU factorization) and then `p?getrs` (computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

[Table “Computational Routines for Systems of Linear Equations”](#) lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	<code>p?getrf</code>	<code>p?geequ</code>	<code>p?getrs</code>	<code>p?gecon</code>	<code>p?gerfs</code>	<code>p?getri</code>
general band (partial pivoting)	<code>p?gbtrf</code>		<code>p?gbtrs</code>			
general band (no pivoting)	<code>p?dbtrf</code>		<code>p?dbtrs</code>			
general tridiagonal (no pivoting)	<code>p?dttrf</code>		<code>p?dttrs</code>			
symmetric/Hermitian positive-definite	<code>p?potrf</code>	<code>p?poequ</code>	<code>p?potrs</code>	<code>p?pocon</code>	<code>p?porfs</code>	<code>p?potri</code>
symmetric/Hermitian positive-definite, band	<code>p?pbtrf</code>		<code>p?pbtrs</code>			
symmetric/Hermitian positive-definite, tridiagonal	<code>p?pttrf</code>		<code>p?pttrs</code>			
triangular			<code>p?trtrs</code>	<code>p?trcon</code>	<code>p?trrfs</code>	<code>p?trtri</code>

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex).

Routines for Matrix Factorization

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

`p?getrf`

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

```
call psgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pdgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pcgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pzgetrf(m, n, a, ia, ja, desca, ipiv, info)
```

Include Files

Description

The `p?getrf` routine forms the LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$ as

$$A = P * L * U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in $\text{sub}(A)$.

The routine uses partial pivoting, with row interchanges.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$; $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$; $n \geq 0$.
<i>a</i>	(local) REAL for <code>psgetrf</code> DOUBLE PRECISION for <code>pdgetrf</code> COMPLEX for <code>pcgetrf</code> DOUBLE COMPLEX for <code>pzgetrf</code> . Pointer into the local memory to an array of local size $(\text{lld}_a, \text{LOCc}(\text{ja} + n - 1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

<i>a</i>	Overwritten by local pieces of the factors L and U from the factorization $A = P * L * U$. The unit diagonal elements of L are not stored.
<i>ipiv</i>	(local) INTEGER Array of size $\text{LOCr}(m_a) + mb_a$.

Contains the pivoting information: local row i was interchanged with global row $ipiv(i)$. This array is tied to the distributed matrix A .

info

(global) INTEGER.

If $info=0$, the execution is successful.

$info < 0$: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info = i > 0$, $u_{ia+i, ja+j-1}$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrf

Computes the LU factorization of a general n -by- n banded distributed matrix.

Syntax

```
call psgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

```
call pdgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

```
call pcgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

```
call pzgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

Include Files

Description

The `p?gbtrf` routine computes the LU factorization of a general n -by- n real/complex banded distributed matrix $A(1:n, ja:ja+n-1)$ using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine `?gbtrf`. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

$$A(1:n, ja:ja+n-1) = P*L*U*Q$$

where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
<i>bwl</i>	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>a</i>	(local) REAL for psgbtrf DOUBLE PRECISION for pdgbtrf COMPLEX for pcgbtrf DOUBLE COMPLEX for pzgbtrf. Pointer into the local memory to an array of local size ($lld_a, LOCC(ja+n-1)$) where $lld_a \geq 2*bwl + 2*bwu + 1$. Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A . If $dtype_a = 501$, then $dlen_ \geq 7$; else if $dtype_a = 1$, then $dlen_ \geq 9$.
<i>laf</i>	(local) INTEGER. The size of the array af . Must be $laf \geq (nb_a+bwu) * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2*bwu)$. If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.
<i>work</i>	(local) Same type as a . Workspace array of size $lwork$.
<i>lwork</i>	(local or global) INTEGER. The size of the $work$ array ($lwork \geq 1$). If $lwork$ is too small, the minimal acceptable size will be returned in $work(1)$ and an error code is returned.

Output Parameters

<i>a</i>	On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>ipiv</i>	(local) INTEGER array. The size of <i>ipiv</i> must be $\geq nb_a$. Contains pivot indices for local factorizations. Note that you <i>should not alter</i> the contents of this array between factorization and solve.
<i>af</i>	(local) REAL for psgbtrf DOUBLE PRECISION for pdgbtrf COMPLEX for pcgbtrf DOUBLE COMPLEX for pzgbtrf. Array of size <i>laf</i> . Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?gbtrf and is stored in <i>af</i> . Note that if a linear system is to be solved using p?gbtrs after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required.
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. <i>info</i> > 0: If <i>info</i> = $k \leq NPROCS$, the submatrix stored on processor <i>info</i> and factored locally was not nonsingular, and the factorization was not completed. If <i>info</i> = $k > NPROCS$, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrf

Computes the LU factorization of a n-by-n diagonally dominant-like banded distributed matrix.

Syntax

```
call psdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pddbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

```
call pcdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

```
call pzdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The `p?dbtrf` routine computes the LU factorization of a n -by- n real/complex diagonally dominant-like banded distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting.

NOTE

A matrix is called *diagonally dominant-like* if pivoting is not required for LU to be numerically stable.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

n	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
bwl	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
a	(local) REAL for <code>psdbtrf</code> DOUBLE PRECISION for <code>pddbtrf</code> COMPLEX for <code>pcdbtrf</code> DOUBLE COMPLEX for <code>pzdbtrf</code> . Pointer into the local memory to an array of local size (<code>lld_a, LOCC(ja+n-1)</code>). Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.

<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) INTEGER. The size of the array <i>af</i> . Must be $laf \geq NB * (bwl + bwu) + 6 * (\max(bwl, bwu))^2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be $lwork \geq (\max(bwl, bwu))^2$. If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned.

Output Parameters

<i>a</i>	On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>af</i>	(local) REAL for <i>psdbtrf</i> DOUBLE PRECISION for <i>pddbtrf</i> COMPLEX for <i>pcdbtrf</i> DOUBLE COMPLEX for <i>pzdbtrf</i> . Array of size <i>laf</i> . Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine <i>p?dbtrf</i> and is stored in <i>af</i> . Note that if a linear system is to be solved using <i>p?dbtrs</i> after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>j</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . <i>info</i> > 0:

If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not diagonally dominant-like, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info - NPROCS$ representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
call psdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pddttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pcdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pzdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The `p?dttrf` routine computes the LU factorization of an n -by- n real/complex diagonally dominant-like tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P * L * U * P^T,$$

where P is a permutation matrix, and L and U are banded lower and upper triangular matrices, respectively.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).

dl, d, du (local)

REAL for `pspttrf`
 DOUBLE PRECISION for `pdpttrf`
 COMPLEX for `pcpttrf`
 DOUBLE COMPLEX for `pzpttrf`.

Pointers to the local arrays of size nb_a each.

On entry, the array dl contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, $dl(1)$ is not referenced, and dl must be aligned with d .

On entry, the array d contains the local part of the global vector storing the diagonal elements of the matrix.

On entry, the array du contains the local part of the global vector storing the super-diagonal elements of the matrix. $du(n)$ is not referenced, and du must be aligned with d .

ja	(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A . If $dtype_a = 501$, then $dlen_ \geq 7$; else if $dtype_a = 1$, then $dlen_ \geq 9$.
laf	(local) INTEGER. The size of the array af . Must be $laf \geq 2*(NB+2)$. If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.
$work$	(local) Same type as d . Workspace array of size $lwork$.
$lwork$	(local or global) INTEGER. The size of the $work$ array, must be at least $lwork \geq 8*NPCOL$.

Output Parameters

dl, d, du	On exit, overwritten by the information containing the factors of the matrix.
af	(local) REAL for <code>psdttrf</code> DOUBLE PRECISION for <code>pddttrf</code> COMPLEX for <code>pcdttrf</code> DOUBLE COMPLEX for <code>pzdttrf</code> . Array of size laf . Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine <code>p?dttrf</code> and is stored in af .

Note that if a linear system is to be solved using `p?dtrtrs` after the factorization routine, `af` must not be altered.

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

If `info=0`, the execution is successful.

`info < 0`:

If the i -th argument is an array and the j -th entry had an illegal value, then `info = -(i*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

`info > 0`:

If `info = k ≤ NPROCS`, the submatrix stored on processor `info` and factored locally was not diagonally dominant-like, and the factorization was not completed.

If `info = k > NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
call pspotrf(uplo, n, a, ia, ja, desca, info)
call pdpotrf(uplo, n, a, ia, ja, desca, info)
call pcpotrf(uplo, n, a, ia, ja, desca, info)
call pzpotrf(uplo, n, a, ia, ja, desca, info)
```

Include Files

Description

The `p?potrf` routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed n -by- n matrix $A(ia:ia+n-1, ja:ja+n-1)$, denoted below as $\text{sub}(A)$.

The factorization has the form

$\text{sub}(A) = U^H * U$ if `uplo='U'`, or

$\text{sub}(A) = L * L^H$ if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

`uplo` (global) CHARACTER*1.

Indicates whether the upper or lower triangular part of sub(A) is stored. Must be 'U' or 'L'.

If $uplo = 'U'$, the array a stores the upper triangular part of the matrix sub(A) that is factored as U^H*U .

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix sub(A) that is factored as $L*L^H$.

n	(global) INTEGER. The order of the distributed matrix sub(A) ($n \geq 0$).
a	(local) REAL for pspotr DOUBLE PRECISION for pdpotrf COMPLEX for pcpotr DOUBLE COMPLEX for pzpotr. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix sub(A) to be factored. Depending on $uplo$, the array a contains either the upper or the lower triangular part of the matrix sub(A) (see $uplo$).
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a	The upper or lower triangular part of a is overwritten by the Cholesky factor U or L , as specified by $uplo$.
$info$	(global) INTEGER. If $info=0$, the execution is successful; $info < 0$: if the i -th argument is an array, and the j -th entry had an illegal value, then $info = -(j*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$. If $info = k > 0$, the leading minor of order k , $A(ia:ia+k-1, ja:ja+k-1)$, is not positive-definite, and the factorization could not be completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
call pspbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pdpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pcpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pzpbtfrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The `p?pbtrf` routine computes the Cholesky factorization of an n -by- n real symmetric or complex Hermitian positive-definite banded distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P*U^H*U*P^T$, if `uplo='U'`, or

$A(1:n, ja:ja+n-1) = P*L*L^H*P^T$, if `uplo='L'`,

where P is a permutation matrix and U and L are banded upper and lower triangular matrices, respectively.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be 'U' or 'L'. If <code>uplo = 'U'</code> , upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <code>uplo = 'L'</code> , lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix $A(1:n, ja:ja+n-1)$. ($n \geq 0$).
<code>bw</code>	(global) INTEGER. The number of superdiagonals of the distributed matrix if <code>uplo = 'U'</code> , or the number of subdiagonals if <code>uplo = 'L'</code> ($bw \geq 0$).
<code>a</code>	(local) REAL for <code>pspbtrf</code> DOUBLE PRECISION for <code>pdpbtrf</code>

COMPLEX for `pcpbtrf`

DOUBLE COMPLEX for `pzpbtrf`.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$.

On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix $A(1:n, ja:ja+n-1)$ to be factored.

ja (global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

If $dtype_a = 501$, then $dlen_ \geq 7$;

else if $dtype_a = 1$, then $dlen_ \geq 9$.

laf (local) INTEGER. The size of the array *af*.

Must be $laf \geq (NB+2*bw) * bw$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.

work (local) Same type as *a*. Workspace array of size *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be $lwork \geq bw^2$.

Output Parameters

a On exit, if $info=0$, contains the permuted triangular factor U or L from the Cholesky factorization of the band matrix $A(1:n, ja:ja+n-1)$, as specified by *uplo*.

af (local)

REAL for `pspbtrf`

DOUBLE PRECISION for `pdpbtrf`

COMPLEX for `pcpbtrf`

DOUBLE COMPLEX for `pzpbtrf`.

Array of size *laf*. Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?pbtrf` and stored in *af*. Note that if a linear system is to be solved using `p?pbtrs` after the factorization routine, *af* must not be altered.

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not positive definite, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info - NPROCS$ representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
call psppttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

```
call pdpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

```
call pcpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

```
call pzpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

Include Files

Description

The `p?pttrf` routine computes the Cholesky factorization of an n -by- n real symmetric or complex hermitian positive-definite tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P * L * D * L^H * P^T, \text{ or}$$

$$A(1:n, ja:ja+n-1) = P * U^H * D * U * P^T,$$

where P is a permutation matrix, and U and L are tridiagonal upper and lower triangular matrices, respectively.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>n</i>	(global) INTEGER. The order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).
<i>d, e</i>	(local) REAL for <code>pspttrf</code> DOUBLE PRECISION for <code>pdpttrf</code> COMPLEX for <code>pcpttrf</code> DOUBLE COMPLEX for <code>pzpttrf</code> . Pointers into the local memory to arrays of size <i>nb_a</i> each. On entry, the array <i>d</i> contains the local part of the global vector storing the main diagonal of the distributed matrix <i>A</i> . On entry, the array <i>e</i> contains the local part of the global vector storing the upper diagonal of the distributed matrix <i>A</i> .
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501, then $dlen_{\geq} 7$; else if <i>dtype_a</i> = 1, then $dlen_{\geq} 9$.
<i>laf</i>	(local) INTEGER. The size of the array <i>af</i> . Must be $laf \geq nb_a + 2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>d</i> and <i>e</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be at least $lwork \geq 8 * NPCOL$.

Output Parameters

<i>d, e</i>	On exit, overwritten by the details of the factorization.
<i>af</i>	(local) REAL for <code>pspttrf</code> DOUBLE PRECISION for <code>pdpttrf</code> COMPLEX for <code>pcpttrf</code> DOUBLE COMPLEX for <code>pzpttrf</code> . Array of size <i>laf</i> .

Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?pttrf` and stored in `af`.

Note that if a linear system is to be solved using `p?pttrs` after the factorization routine, `af` must not be altered.

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

If `info=0`, the execution is successful.

`info < 0`:

If the i -th argument is an array and the j -th entry had an illegal value, then `info = -(j*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

`info > 0`:

If `info = k ≤ NPROCS`, the submatrix stored on processor `info` and factored locally was not positive definite, and the factorization was not completed.

If `info = k > NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Routines for Solving Systems of Linear Equations

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

```
call psgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

```
call pdgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

```
call pcgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

```
call pzgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Include Files

Description

The `p?getrs` routine solves a system of distributed linear equations with a general n -by- n distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the LU factorization computed by `p?getrf`.

The system has one of the following forms specified by `trans`:

$\text{sub}(A)*X = \text{sub}(B)$ (no transpose),
 $\text{sub}(A)^T*X = \text{sub}(B)$ (transpose),
 $\text{sub}(A)^H*X = \text{sub}(B)$ (conjugate transpose),
 where $\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1)$.

Before calling this routine, you must call `p?getrf` to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations:
 If *trans* = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X .
 If *trans* = 'T', then $\text{sub}(A)^T*X = \text{sub}(B)$ is solved for X .
 If *trans* = 'C', then $\text{sub}(A)^H *X = \text{sub}(B)$ is solved for X .

n (global) INTEGER. The number of linear equations; the order of the matrix $\text{sub}(A)$ ($n \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).

a, b (global)
 REAL for `psgetrs`
 DOUBLE PRECISION for `pdgetrs`
 COMPLEX for `pcgetrs`
 DOUBLE COMPLEX for `pzgetrs`.
 Pointers into the local memory to arrays of local sizes $(lld_a, LOCc(ja+n-1))$ and $(lld_b, LOCc(jb+nrhs-1))$, respectively.
 On entry, the array *a* contains the local pieces of the factors L and U from the factorization $\text{sub}(A) = P*L*U$; the unit diagonal elements of L are not stored. On entry, the array *b* contains the right hand sides $\text{sub}(B)$.

ia, ja (global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

ipiv (local) INTEGER Array of size of $LOCr(m_a) + mb_a$. Contains the pivoting information: local row i of the matrix was interchanged with the global row $ipiv(i)$.
 This array is tied to the distributed matrix A .

ib, jb (global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

b On exit, overwritten by the solution distributed matrix *X*.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*j**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

```
call psgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pdgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pcgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pzgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)
```

Include Files

Description

The `p?gbtrs` routine solves a system of distributed linear equations with a general band distributed matrix $\text{sub}(A) = A(1:n, ja:ja+n-1)$ using the *LU* factorization computed by `p?gbtrf`.

The system has one of the following forms specified by *trans*:

$\text{sub}(A)*X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T*X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H*X = \text{sub}(B)$ (conjugate transpose),

where $\text{sub}(B) = B(ib:ib+n-1, 1:nrhs)$.

Before calling this routine, you must call `p?gbtrf` to compute the *LU* factorization of $\text{sub}(A)$.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Indicates the form of the equations:
 If *trans* = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for *X*.

	<p>If $trans = 'T'$, then $sub(A)^T * X = sub(B)$ is solved for X.</p> <p>If $trans = 'C'$, then $sub(A)^H * X = sub(B)$ is solved for X.</p>
<i>n</i>	(global) INTEGER. The number of linear equations; the order of the distributed matrix $sub(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $sub(B)$ ($nrhs \geq 0$).
<i>a, b</i>	<p>(global)</p> <p>REAL for <code>psgbtrs</code></p> <p>DOUBLE PRECISION for <code>pdgbtrs</code></p> <p>COMPLEX for <code>pcgbtrs</code></p> <p>DOUBLE COMPLEX for <code>pzgbtrs</code>.</p> <p>Pointers into the local memory to arrays of local sizes ($lld_a, LOCC(ja + n - 1)$) and ($lld_b, LOCC(nrhs)$), respectively.</p> <p>The array <i>a</i> contains details of the LU factorization of the distributed band matrix <i>A</i>.</p> <p>On entry, the array <i>b</i> contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.</p>
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If $dtype_a = 501$, then $dlen_ \geq 7$;</p> <p>else if $dtype_a = 1$, then $dlen_ \geq 9$.</p>
<i>ib</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If $dtype_b = 502$, then $dlen_ \geq 7$;</p> <p>else if $dtype_b = 1$, then $dlen_ \geq 9$.</p>
<i>laf</i>	<p>(local) INTEGER. The size of the array <i>af</i>.</p> <p>Must be $laf \geq nb_a * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$.</p>

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work (local) Same type as *a*. Workspace array of size *lwork*.

lwork (local or global) INTEGER. The size of the *work* array, must be at least $lwork \geq nrhs * (nb_a + 2 * bwl + 4 * bwu)$.

Output Parameters

ipiv (local) INTEGER array.

The size of *ipiv* must be $\geq nb_a$.

Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.

b On exit, overwritten by the local pieces of the solution distributed matrix X.

af (local)

REAL for psgbtrs

DOUBLE PRECISION for pdgbtrs

COMPLEX for pcgbtrs

DOUBLE COMPLEX for pzgbtrs.

Array of size *laf*.

Auxiliary Fill-in space. The fill-in space is created in a call to the factorization routine *p?gbtrf* and is stored in *af*.

Note that if a linear system is to be solved using *p?gbtrs* after the factorization routine, *af* must not be altered after the factorization.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = $-(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by p?dbtrf.

Syntax

```
call psdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pddbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pcdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pzdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

Include Files

Description

The `p?dbtrs` routine solves for X one of the systems of equations:

$$\text{sub}(A)*X = \text{sub}(B),$$

$$(\text{sub}(A))^T*X = \text{sub}(B), \text{ or}$$

$$(\text{sub}(A))^H*X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like banded distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by `p?dbtrf`.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'T', then $(\text{sub}(A))^T*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'C', then $(\text{sub}(A))^H*X = \text{sub}(B)$ is solved for X .
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for <code>psdbtrs</code> DOUBLE PRECISION for <code>pddbtrs</code> COMPLEX for <code>pcdbtrs</code> DOUBLE COMPLEX for <code>pzdbtrs</code> . Pointers into the local memory to arrays of local sizes (<code>lld_a, LOCC(ja+n-1)</code>) and (<code>lld_b, LOCC(nrhs)</code>), respectively.

On entry, the array *a* contains details of the *LU* factorization of the band matrix *A*, as computed by `p?dbtrf`.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix `sub(B)`.

ja (global) INTEGER. The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

ib (global) INTEGER. The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *dtype_b* = 502, then *dlen_* ≥ 7;

else if *dtype_b* = 1, then *dlen_* ≥ 9.

af, work

(local)

REAL for `psdbtrs`

DOUBLE PRECISION for `pddbtrs`

COMPLEX for `pcdbtrs`

DOUBLE COMPLEX for `pzdbtrs`.

Arrays of size *laf* and *lwork*, respectively. The array *af* contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?dbtrf` and is stored in *af*.

The array *work* is a workspace array.

laf (local) INTEGER. The size of the array *af*.

Must be $laf \geq NB * (bwl + bwu) + 6 * (\max(bwl, bwu))^2$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least

$lwork \geq (\max(bwl, bwu))^2$.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

```
call psdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pddttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pcdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pzdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

Include Files

Description

The p?dttrs routine solves for X one of the systems of equations:

$$\text{sub}(A)*X = \text{sub}(B),$$

$$(\text{sub}(A))^T*X = \text{sub}(B), \text{ or}$$

$$(\text{sub}(A))^H*X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by p?dttrf.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X .

If *trans* = 'T', then $(\text{sub}(A))^T*X = \text{sub}(B)$ is solved for X .

If *trans* = 'C', then $(\text{sub}(A))^H*X = \text{sub}(B)$ is solved for X .

n (global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).

<i>dl, d, du</i>	<p>(local)</p> <p>REAL for psdttrs</p> <p>DOUBLE PRECISION for pddttrs</p> <p>COMPLEX for pcdttrs</p> <p>DOUBLE COMPLEX for pzdttrs.</p> <p>Pointers to the local arrays of size <i>nb_a</i> each.</p> <p>On entry, these arrays contain details of the factorization. Globally, <i>dl(1)</i> and <i>du(n)</i> are not referenced; <i>dl</i> and <i>du</i> must be aligned with <i>d</i>.</p>
<i>ja</i>	<p>(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).</p>
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If <i>dtype_a</i> = 501 or <i>dtype_a</i> = 502, then <i>dlen_</i> ≥ 7;</p> <p>else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>b</i>	<p>(local) Same type as <i>d</i>.</p> <p>Pointer into the local memory to an array of local size (<i>lld_b, LOcc(nrhs)</i>)</p> <p>On entry, the array <i>b</i> contains the local pieces of the <i>n</i>-by-<i>nrhs</i> right hand side distributed matrix sub(<i>B</i>).</p>
<i>ib</i>	<p>(global) INTEGER. The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).</p>
<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>.</p> <p>If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7;</p> <p>else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>af, work</i>	<p>(local) REAL for psdttrs</p> <p>DOUBLE PRECISION for pddttrs</p> <p>COMPLEX for pcdttrs</p> <p>DOUBLE COMPLEX for pzdttrs.</p> <p>Arrays of size <i>laf</i> and (<i>lwork</i>), respectively.</p> <p>The array <i>af</i> contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine <i>p?dttrf</i> and is stored in <i>af</i>. If a linear system is to be solved using <i>p?dttrs</i> after the factorization routine, <i>af</i> must not be altered.</p> <p>The array <i>work</i> is a workspace array.</p>
<i>laf</i>	<p>(local) INTEGER. The size of the array <i>af</i>.</p>

Must be $laf \geq NB * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$.

If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.

lwork (local or global) INTEGER. The size of the array *work*, must be at least $lwork \geq 10 * NPCOL + 4 * nrhs$.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix X .

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. $info < 0$:
If the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(j * 100 + i)$; if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
call pspotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzpotsrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Include Files

Description

The p?potrs routine solves for X a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

This routine uses Cholesky factorization

$$\text{sub}(A) = U^H * U, \text{ or } \text{sub}(A) = L * L^H$$

computed by p?potrf.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

	<p>If <code>uplo = 'U'</code>, upper triangle of <code>sub(A)</code> is stored;</p> <p>If <code>uplo = 'L'</code>, lower triangle of <code>sub(A)</code> is stored.</p>
<code>n</code>	(global) INTEGER. The order of the distributed matrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix <code>sub(B)</code> ($nrhs \geq 0$).
<code>a, b</code>	<p>(local)</p> <p>REAL for <code>pspotrs</code></p> <p>DOUBLE PRECISION for <code>pdpotrs</code></p> <p>COMPLEX for <code>pcpotrs</code></p> <p>DOUBLE COMPLEX for <code>pzpotrs</code>.</p> <p>Pointers into the local memory to arrays of local sizes $(lld_a, LOcc(ja+n-1))$ and $(lld_b, LOcc(jb+nrhs-1))$, respectively.</p> <p>The array <code>a</code> contains the factors <code>L</code> or <code>U</code> from the Cholesky factorization $sub(A) = L * L^H$ or $sub(A) = U^H * U$, as computed by <code>p?potrf</code>.</p> <p>On entry, the array <code>b</code> contains the local pieces of the right hand sides <code>sub(B)</code>.</p>
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global matrix <code>A</code> indicating the first row and the first column of the matrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>A</code> .
<code>ib, jb</code>	(global) INTEGER. The row and column indices in the global matrix <code>B</code> indicating the first row and the first column of the matrix <code>sub(B)</code> , respectively.
<code>descb</code>	(local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>B</code> .

Output Parameters

<code>b</code>	Overwritten by the local pieces of the solution matrix <code>X</code> .
<code>info</code>	<p>INTEGER. If <code>info=0</code>, the execution is successful.</p> <p><code>info < 0</code>: if the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <code>info = -(i*100+j)</code>; if the <i>i</i>-th argument is a scalar and had an illegal value, then <code>info = -i</code>.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
call pspbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pdpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pcpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pzpbttrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The `p?pbttrs` routine solves for X a system of distributed linear equations in the form:

$$\text{sub}(A)*X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed band matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses Cholesky factorization

$$\text{sub}(A) = P*U^H*U*P^T, \text{ or } \text{sub}(A) = P*L*L^H*P^T$$

computed by `p?pbttrf`.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $\text{sub}(A)$ is stored; If <i>uplo</i> = 'L', lower triangle of $\text{sub}(A)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>bw</i>	(global) INTEGER. The number of superdiagonals of the distributed matrix if <i>uplo</i> = 'U', or the number of subdiagonals if <i>uplo</i> = 'L' ($bw \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for <code>pspbtrs</code> DOUBLE PRECISION for <code>pdpbtrs</code> COMPLEX for <code>pcpbtrs</code> DOUBLE COMPLEX for <code>pzpbttrs</code> . Pointers into the local memory to arrays of local sizes (<code>lld_a, LOCC(ja+n-1)</code>) and (<code>lld_b, LOCC(nrhs-1)</code>), respectively. The array <i>a</i> contains the permuted triangular factor U or L from the Cholesky factorization $\text{sub}(A) = P*U^H*U*P^T$, or $\text{sub}(A) = P*L*L^H*P^T$ of the band matrix A , as returned by <code>p?pbttrf</code> .

	On entry, the array <i>b</i> contains the local pieces of the <i>n</i> -by- <i>nrhs</i> right hand side distributed matrix <i>sub(B)</i> .
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.
<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> indicating the first row of the matrix <i>sub(B)</i> .
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.
<i>af, work</i>	(local) Arrays, same type as <i>a</i> . The array <i>af</i> is of size <i>laf</i> . It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p?dbtrf and is stored in <i>af</i> . The array <i>work</i> is a workspace array of size <i>lwork</i> .
<i>laf</i>	(local) INTEGER. The size of the array <i>af</i> . Must be <i>laf</i> ≥ <i>nrhs</i> * <i>bw</i> . If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> , must be at least <i>lwork</i> ≥ <i>bw</i> ² .

Output Parameters

<i>b</i>	On exit, if <i>info</i> =0, this array contains the local pieces of the <i>n</i> -by- <i>nrhs</i> solution distributed matrix <i>X</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>j</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf.

Syntax

```
call pspttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Include Files

Description

The p?pttrs routine solves for X a system of distributed linear equations in the form:

$$\text{sub}(A)*X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the factorization

$$\text{sub}(A) = P*L*D*L^H*P^T, \text{ or } \text{sub}(A) = P*U^H*D*U*P^T$$

computed by p?pttrf.

Input Parameters

<i>uplo</i>	(global, used in complex flavors only) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $\text{sub}(A)$ is stored; If <i>uplo</i> = 'L', lower triangle of $\text{sub}(A)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>d, e</i>	(local) REAL for pspttrs DOUBLE PRECISION for pdpttrs COMPLEX for pcpttrs DOUBLE COMPLEX for pzpttrs. Pointers into the local memory to arrays of size <i>nb_a</i> each. These arrays contain details of the factorization as returned by p?pttrf
<i>ja</i>	(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If <i>dtype_a</i> = 501 or <i>dtype_a</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>b</i>	<p>(local) Same type as <i>d</i>, <i>e</i>.</p> <p>Pointer into the local memory to an array of local size <i>(lld_b, LOCC(nrhs))</i>.</p> <p>On entry, the array <i>b</i> contains the local pieces of the <i>n</i>-by-<i>nrhs</i> right hand side distributed matrix sub(<i>B</i>).</p>
<i>ib</i>	<p>(global) INTEGER. The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).</p>
<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>.</p> <p>If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>af, work</i>	<p>(local) REAL for <i>pspttrs</i> DOUBLE PRECISION for <i>pdpttrs</i> COMPLEX for <i>pcpttrs</i> DOUBLE COMPLEX for <i>pzpttrs</i>.</p> <p>Arrays of size <i>laf</i> and (<i>lwork</i>), respectively. The array <i>af</i> contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine <i>p?pttrf</i> and is stored in <i>af</i>.</p> <p>The array <i>work</i> is a workspace array.</p>
<i>laf</i>	<p>(local) INTEGER. The size of the array <i>af</i>.</p> <p>Must be <i>laf</i> ≥ <i>nb_a</i> + 2.</p> <p>If <i>laf</i> is not large enough, an error code is returned and the minimum acceptable size will be returned in <i>af</i>(1).</p>
<i>lwork</i>	<p>(local or global) INTEGER. The size of the array <i>work</i>, must be at least <i>lwork</i> ≥ (10 + 2 * min(100, <i>nrhs</i>)) * NPCOL + 4 * <i>nrhs</i>.</p>

Output Parameters

<i>b</i>	<p>On exit, this array contains the local pieces of the solution distributed matrix <i>X</i>.</p>
<i>work</i> (1)	<p>On exit, <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance.</p>
<i>info</i>	<p>INTEGER. If <i>info</i> = 0, the execution is successful.</p> <p><i>info</i> < 0:</p>

if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

```
call pstrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

```
call pdtrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

```
call pctrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

```
call pztrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Include Files

Description

The p?trtrs routine solves for X one of the following systems of linear equations:

$\text{sub}(A)*X = \text{sub}(B)$,

$(\text{sub}(A))^T*X = \text{sub}(B)$, or

$(\text{sub}(A))^H*X = \text{sub}(B)$,

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a triangular distributed matrix of order n , and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

A check is made to verify that $\text{sub}(A)$ is nonsingular.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Indicates whether $\text{sub}(A)$ is upper or lower triangular: If <i>uplo</i> = 'U', then $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', then $\text{sub}(A)$ is lower triangular.
<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'T', then $(\text{sub}(A))^T*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'C', then $(\text{sub}(A))^H*X = \text{sub}(B)$ is solved for X .
<i>diag</i>	(global) CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $\text{sub}(A)$ is not a unit triangular matrix. If <i>diag</i> = 'U', then $\text{sub}(A)$ is unit triangular.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).

<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; i.e., the number of columns of the distributed matrix sub(<i>B</i>) (<i>nrhs</i> ≥0).
<i>a, b</i>	<p>(local)</p> <p>REAL for pstrtrs</p> <p>DOUBLE PRECISION for pdtrtrs</p> <p>COMPLEX for pctrtrs</p> <p>DOUBLE COMPLEX for pztrtrs.</p> <p>Pointers into the local memory to arrays of local sizes (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i> + <i>n</i> - 1)) and (<i>lld_b</i>, <i>LOCc</i>(<i>jb</i> + <i>nrhs</i> - 1)), respectively.</p> <p>The array <i>a</i> contains the local pieces of the distributed triangular matrix sub(<i>A</i>).</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular matrix, and the strictly lower triangular part of sub(<i>A</i>) is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular matrix, and the strictly upper triangular part of sub(<i>A</i>) is not referenced.</p> <p>If <i>diag</i> = 'U', the diagonal elements of sub(<i>A</i>) are also not referenced and are assumed to be 1.</p> <p>On entry, the array <i>b</i> contains the local pieces of the right hand side distributed matrix sub(<i>B</i>).</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	On exit, if <i>info</i> =0, sub(<i>B</i>) is overwritten by the solution matrix <i>X</i> .
<i>info</i>	<p>INTEGER. If <i>info</i>=0, the execution is successful.</p> <p><i>info</i> < 0:</p> <p>if the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>); if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p> <p><i>info</i> > 0:</p>

if $info = i$, the i -th diagonal element of $sub(A)$ is zero, indicating that the submatrix is singular and the solutions X have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Routines for Estimating the Condition Number

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
call psgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
```

```
call pdgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork, info)
```

```
call pcgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
```

```
call pzgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The `p?gecon` routine estimates the reciprocal of the condition number of a general distributed real/complex matrix $sub(A) = A(ia:ia+n-1, ja:ja+n-1)$ in either the 1-norm or infinity-norm, using the LU factorization computed by `p?getrf`.

An estimate is obtained for $|(sub(A))^{-1}|$, and the reciprocal of the condition number is computed as

```
.....!..... !.....! .....
```

Input Parameters

`norm` (global) CHARACTER*1. Must be '1' or 'O' or 'I'.
 Specifies whether the 1-norm condition number or the infinity-norm condition number is required.
 If `norm = '1'` or 'O', then the 1-norm is used;
 If `norm = 'I'`, then the infinity-norm is used.

<i>n</i>	(global) INTEGER. The order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>REAL for psgecon</p> <p>DOUBLE PRECISION for pdgecon</p> <p>COMPLEX for pcgecon</p> <p>DOUBLE COMPLEX for pzgecon.</p> <p>Pointer into the local memory to an array of size (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)).</p> <p>The array <i>a</i> contains the local pieces of the factors <i>L</i> and <i>U</i> from the factorization $\text{sub}(A) = P*L*U$; the unit diagonal elements of <i>L</i> are not stored.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>anorm</i>	<p>(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors.</p> <p>If <i>norm</i> = '1' or 'O', the 1-norm of the original distributed matrix sub(<i>A</i>);</p> <p>If <i>norm</i> = 'I', the infinity-norm of the original distributed matrix sub(<i>A</i>).</p>
<i>work</i>	<p>(local)</p> <p>REAL for psgecon</p> <p>DOUBLE PRECISION for pdgecon</p> <p>COMPLEX for pcgecon</p> <p>DOUBLE COMPLEX for pzgecon.</p> <p>The array <i>work</i> of size <i>lwork</i> is a workspace array.</p>
<i>lwork</i>	<p>(local or global) INTEGER. The size of the array <i>work</i>.</p> <p>For real flavors:</p> <p><i>lwork</i> must be at least</p> $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a))+2*LOCc(n+\text{mod}(ja-1, nb_a)) + \max(2, \max(nb_a*\max(1, \text{ceil}(\text{NPROW}-1, NPCOL)), LOCc(n+\text{mod}(ja-1, nb_a)) + nb_a*\max(1, \text{ceil}(\text{NPCOL}-1, \text{NPROW}))))).$ <p>For complex flavors:</p> <p><i>lwork</i> must be at least</p> $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a))+\max(2, \max(nb_a*\text{ceil}(\text{NPROW}-1, NPCOL), LOCc(n+\text{mod}(ja-1, nb_a))+nb_a*\text{ceil}(\text{NPCOL}-1, \text{NPROW}))).$ <p><i>LOCr</i> and <i>LOCc</i> values can be computed using the ScaLAPACK tool function numroc; <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine blacs_gridinfo.</p>

NOTE

$\text{iceil}(x, y)$ is the ceiling of x/y , and $\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ia-1, mb_a))$.
<i>rwork</i>	(local) REAL for <i>pcgecon</i> DOUBLE PRECISION for <i>pzgecon</i> Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq \max(1, 2 * LOCc(n + \text{mod}(ja-1, nb_a)))$.

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The reciprocal of the condition number of the distributed matrix sub(A). See Description.
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>j</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>anorm</i>	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. The 1-norm of the symmetric/Hermitian distributed matrix sub(<i>A</i>).
<i>work</i>	(local) REAL for <i>pspocon</i> DOUBLE PRECISION for <i>pdpocon</i> COMPLEX for <i>pcpocon</i> DOUBLE COMPLEX for <i>pzpocon</i> . The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a)) + 2*LOCc(n+\text{mod}(ja-1, nb_a)) + \max(2, \max(nb_a*\text{ceil}(NPROW-1, NPCOL), LOCc(n+\text{mod}(ja-1, nb_a)) + nb_a*\text{ceil}(NPCOL-1, NPROW)))$ <i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a)) + \max(2, \max(nb_a*\max(1, \text{ceil}(NPROW-1, NPCOL)), LOCc(n+\text{mod}(ja-1, nb_a)) + nb_a*\max(1, \text{ceil}(NPCOL-1, NPROW))))$
<hr/> NOTE ceil(<i>x</i> , <i>y</i>) is the ceiling of <i>x</i> / <i>y</i> , and mod(<i>x</i> , <i>y</i>) is the integer remainder of <i>x</i> / <i>y</i> . <hr/>	
<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n+\text{mod}(ia-1, mb_a))$.
<i>rwork</i>	(local) REAL for <i>pcpocon</i> DOUBLE PRECISION for <i>pzpocon</i> Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq 2*LOCc(n+\text{mod}(ja-1, nb_a))$.

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors.
--------------	---

Input Parameters

<i>norm</i>	<p>(global) CHARACTER*1. Must be '1' or 'O' or 'I'.</p> <p>Specifies whether the 1-norm condition number or the infinity-norm condition number is required.</p> <p>If <i>norm</i> = '1' or 'O', then the 1-norm is used;</p> <p>If <i>norm</i> = 'I', then the infinity-norm is used.</p>
<i>uplo</i>	<p>(global) CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', sub(A) is upper triangular. If <i>uplo</i> = 'L', sub(A) is lower triangular.</p>
<i>diag</i>	<p>(global) CHARACTER*1. Must be 'N' or 'U'.</p> <p>If <i>diag</i> = 'N', sub(A) is non-unit triangular. If <i>diag</i> = 'U', sub(A) is unit triangular.</p>
<i>n</i>	(global) INTEGER. The order of the distributed matrix sub(A), ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>REAL for pstrcon</p> <p>DOUBLE PRECISION for pdtrcon</p> <p>COMPLEX for pctrcon</p> <p>DOUBLE COMPLEX for pztrcon.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$.</p> <p>The array <i>a</i> contains the local pieces of the triangular distributed matrix sub(A).</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.</p> <p>If <i>diag</i> = 'U', the diagonal elements of sub(A) are also not referenced and are assumed to be 1.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	<p>(local)</p> <p>REAL for pstrcon</p> <p>DOUBLE PRECISION for pdtrcon</p> <p>COMPLEX for pctrcon</p> <p>DOUBLE COMPLEX for pztrcon.</p>

The array *work* of size *lwork* is a workspace array.

lwork

(local or global) INTEGER. The size of the array *work*.

For real flavors:

lwork must be at least

$$lwork \geq 2 * LOCr(n + \text{mod}(ia - 1, mb_a)) + LOCc(n + \text{mod}(ja - 1, nb_a)) + \max(2, \max(nb_a * \max(1, \text{ceil}(\text{NPROW} - 1, \text{NPCOL})), LOCc(n + \text{mod}(ja - 1, nb_a)) + nb_a * \max(1, \text{ceil}(\text{NPCOL} - 1, \text{NPROW}))))).$$

For complex flavors:

lwork must be at least

$$lwork \geq 2 * LOCr(n + \text{mod}(ia - 1, mb_a)) + \max(2, \max(nb_a * \text{ceil}(\text{NPROW} - 1, \text{NPCOL}), LOCc(n + \text{mod}(ja - 1, nb_a)) + nb_a * \text{ceil}(\text{NPCOL} - 1, \text{NPROW}))).$$

NOTE

$\text{ceil}(x, y)$ is the ceiling of x/y , and $\text{mod}(x, y)$ is the integer remainder of x/y .

iwork

(local) INTEGER. Workspace array of size *liwork*. Used in real flavors only.

liwork

(local or global) INTEGER. The size of the array *iwork*; used in real flavors only. Must be at least

$$liwork \geq LOCr(n + \text{mod}(ia - 1, mb_a)).$$

rwork

(local) REAL for pcpocon

DOUBLE PRECISION for pzpcocon

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork

(local or global) INTEGER. The size of the array *rwork*; used in complex flavors only. Must be at least

$$lrwork \geq LOCc(n + \text{mod}(ja - 1, nb_a)).$$

Output Parameters

rcond

(global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The reciprocal of the condition number of the distributed matrix sub(A).

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)

On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1)

On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.
info < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Refining the Solution and Estimating Its Error

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Solving Systems of Linear Equations](#)).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
call psgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pdgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pcgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

```
call pzgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The p?gerfs routine improves the computed solution to one of the systems of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

$$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B), \text{ or}$$

$$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B) \text{ and provides error bounds and backward error estimates for the solution.}$$

Here $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+\text{nrhs}-1)$, and $\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+\text{nrhs}-1)$.

Input Parameters

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Specifies the form of the system of equations:
 If *trans* = 'N', the system has the form $\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$ (No transpose);

	<p>If <i>trans</i> = 'T', the system has the form $\text{sub}(A)^T \text{sub}(X) = \text{sub}(B)$ (Transpose);</p> <p>If <i>trans</i> = 'C', the system has the form $\text{sub}(A)^H \text{sub}(X) = \text{sub}(B)$ (Conjugate transpose).</p>
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, af, b, x</i>	<p>(local)</p> <p>REAL for <i>psgerfs</i></p> <p>DOUBLE PRECISION for <i>pdgerfs</i></p> <p>COMPLEX for <i>pcgerfs</i></p> <p>DOUBLE COMPLEX for <i>pzgerfs</i>.</p> <p>Pointers into the local memory to arrays of local sizes <i>a</i> (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i> + <i>n</i> - 1)), <i>af</i> (<i>lld_af</i>, <i>LOCc</i>(<i>jaf</i> + <i>n</i> - 1)), <i>b</i> (<i>lld_b</i>, <i>LOCc</i>(<i>jb</i> + <i>nrhs</i> - 1)), and <i>x</i> (<i>lld_x</i>, <i>LOCc</i>(<i>jx</i> + <i>nrhs</i> - 1)), respectively.</p> <p>The array <i>a</i> contains the local pieces of the distributed matrix $\text{sub}(A)$.</p> <p>The array <i>af</i> contains the local pieces of the distributed factors of the matrix $\text{sub}(A) = P * L * U$ as computed by <i>p?getrf</i>.</p> <p>The array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.</p> <p>On entry, the array <i>x</i> contains the local pieces of the distributed solution matrix $\text{sub}(X)$.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global matrix <i>AF</i> indicating the first row and the first column of the matrix $\text{sub}(AF)$, respectively.
<i>descaf</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.

<i>descx</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>ipiv</i>	(local) INTEGER. Array of size $LOCr(m_af) + mb_af$. This array contains pivoting information as computed by <code>p?getrf</code> . If $ipiv(i)=j$, then the local row <i>i</i> was swapped with the global row <i>j</i> . This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <code>psgerfs</code> DOUBLE PRECISION for <code>pdgerfs</code> COMPLEX for <code>pcgerfs</code> DOUBLE COMPLEX for <code>pzgerfs</code> . The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ <i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$

NOTE
 $\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) INTEGER. Workspace array, size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.
<i>rwork</i>	(local) REAL for <code>pcgerfs</code> DOUBLE PRECISION for <code>pzgerfs</code> Workspace array, size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr, berr</i>	REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size $LOCc(jb+nrhs-1)$ each.

The array *ferr* contains the estimated forward error bound for each solution vector of sub(*X*).

If XTRUE is the true solution corresponding to sub(*X*), *ferr* is an estimated upper bound for the magnitude of the largest element in (sub(*X*) - XTRUE) divided by the magnitude of the largest element in sub(*X*). The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix *X*.

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(*A*) or sub(*B*) that makes sub(*X*) an exact solution). This array is tied to the distributed matrix *X*.

<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>j</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

```
call psporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pdporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pcporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

```
call pzporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The `p?porfs` routine improves the computed solution to the system of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a real symmetric or complex Hermitian positive definite distributed matrix and

$$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1),$$

$$\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+nrhs-1)$$

are right-hand side and solution submatrices, respectively. This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, af, b, x</i>	(local) REAL for <code>psporfs</code> DOUBLE PRECISION for <code>pdporfs</code> COMPLEX for <code>pcporfs</code> DOUBLE COMPLEX for <code>pzporfs</code> . Pointers into the local memory to arrays of local sizes $a(\text{lld}_a, \text{LOCc}(\text{ja}+n-1))$, $af(\text{lld}_{af}, \text{LOCc}(\text{jaf}+n-1))$, $b(\text{lld}_b, \text{LOCc}(\text{jb}+nrhs-1))$, and $x(\text{lld}_x, \text{LOCc}(\text{jx}+nrhs-1))$, respectively. The array <i>a</i> contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. The array <i>af</i> contains the factors L or U from the Cholesky factorization $\text{sub}(A) = L * L^H$ or $\text{sub}(A) = U^H * U$, as computed by <code>p?potrf</code> . On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.

	On entry, the array x contains the local pieces of the solution vectors $\text{sub}(X)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
iaf, jaf	(global) INTEGER. The row and column indices in the global matrix AF indicating the first row and the first column of the matrix $\text{sub}(AF)$, respectively.
$descaf$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix AF .
ib, jb	(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.
$descb$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .
ix, jx	(global) INTEGER. The row and column indices in the global matrix X indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.
$descx$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix X .
$work$	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. The array $work$ of size $lwork$ is a workspace array.
$lwork$	(local) INTEGER. The size of the array $work$. <i>For real flavors:</i> $lwork$ must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ <i>For complex flavors:</i> $lwork$ must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.
<i>rwork</i>	(local) REAL for pcporfs DOUBLE PRECISION for pzporfs Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays of size $LOCc(jb + nrhs - 1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of sub(<i>X</i>). If XTRUE is the true solution corresponding to sub(<i>X</i>), <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in (sub(<i>X</i>) - XTRUE) divided by the magnitude of the largest element in sub(<i>X</i>). The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix <i>X</i> . The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(<i>A</i>) or sub(<i>B</i>) that makes sub(<i>X</i>) an exact solution). This array is tied to the distributed matrix <i>X</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then $info = -(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

```
call pstrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
            jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pdtrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
            jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```
call pctrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
            jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

```
call pztrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
            jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The `p?trrfs` routine provides error bounds and backward error estimates for the solution to one of the systems of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

$$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B), \text{ or}$$

$$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a triangular matrix,

$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$, and

$\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+nrhs-1)$.

The solution matrix X must be computed by `p?trtrs` or some other means before entering this routine. The routine `p?trrfs` does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$ (No transpose); If <i>trans</i> = 'T', the system has the form $\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B)$ (Transpose); If <i>trans</i> = 'C', the system has the form $\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B)$ (Conjugate transpose).
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then $\text{sub}(A)$ is non-unit triangular.

	If <i>diag</i> = 'U', then sub(<i>A</i>) is unit triangular.
<i>n</i>	(global) INTEGER. The order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides, that is, the number of columns of the matrices sub(<i>B</i>) and sub(<i>X</i>) ($nrhs \geq 0$).
<i>a, b, x</i>	(local) REAL for pstrrfs DOUBLE PRECISION for pdtrrfs COMPLEX for pctrfrfs DOUBLE COMPLEX for pztrrfs. Pointers into the local memory to arrays of local sizes <i>a</i> (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> - 1)), <i>b</i> (<i>lld_b</i> , <i>LOCc</i> (<i>jb</i> + <i>nrhs</i> - 1)), and <i>x</i> (<i>lld_x</i> , <i>LOCc</i> (<i>jx</i> + <i>nrhs</i> - 1)), respectively. The array <i>a</i> contains the local pieces of the original triangular distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. If <i>diag</i> = 'U', the diagonal elements of sub(<i>A</i>) are also not referenced and are assumed to be 1. On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides sub(<i>B</i>). On entry, the array <i>x</i> contains the local pieces of the solution vectors sub(<i>X</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the matrix sub(<i>X</i>), respectively.

<i>descx</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	(local) REAL for <i>pstrrfs</i> DOUBLE PRECISION for <i>pdtrrfs</i> COMPLEX for <i>pctrfrfs</i> DOUBLE COMPLEX for <i>pztrrfs</i> . The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local) INTEGER. The size of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$
<hr/> NOTE $\text{mod}(x, y)$ is the integer remainder of x/y . <hr/>	
<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.
<i>rwork</i>	(local) REAL for <i>pctrfrfs</i> DOUBLE PRECISION for <i>pztrrfs</i> Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays of size $LOCc(jb + nrhs - 1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of $\text{sub}(X)$. If <i>XTRUE</i> is the true solution corresponding to $\text{sub}(X)$, <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - XTRUE)$ divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
-------------------	---

This array is tied to the distributed matrix X .

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix X .

<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Routines for Matrix Inversion

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
call psgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pdgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pcgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pzgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
```

Include Files

Description

The *p?getri* routine computes the inverse of a general distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ using the *LU* factorization computed by *p?getrf*. This method inverts *U* and then computes the inverse of $\text{sub}(A)$ by solving the system

$$\text{inv}(\text{sub}(A)) * L = \text{inv}(U)$$

for $\text{inv}(\text{sub}(A))$.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>a</i>	(local) REAL for psgetri DOUBLE PRECISION for pdgetri COMPLEX for pcgetri DOUBLE COMPLEX for pzgetri. Pointer into the local memory to an array of local size ($lld_a, LOCc(ja + n - 1)$). On entry, the array <i>a</i> contains the local pieces of the <i>L</i> and <i>U</i> obtained by the factorization $sub(A) = P * L * U$ computed by p?getrf.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgetri DOUBLE PRECISION for pdgetri COMPLEX for pcgetri DOUBLE COMPLEX for pzgetri. The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local) INTEGER. The size of the array <i>work</i> . <i>lwork</i> must be at least $lwork \geq LOCr(n + \text{mod}(ia - 1, mb_a)) * nb_a$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

The array *work* is used to keep at most an entire column block of sub(*A*).

iwork (local) INTEGER. Workspace array used for physically transposing the pivots, size *liwork*.

liwork (local or global) INTEGER. The size of the array *iwork*.

The minimal value *liwork* of is determined by the following code:

```

if NPROW == NPCOL then
  liwork = LOCC(n_a + mod(ja-1,nb_a)) + nb_a
else
  liwork = LOCC(n_a + mod(ja-1,nb_a)) +
  max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
end if

```

where *lcm* is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

<i>ipiv</i>	<p>(local) INTEGER.</p> <p>Array of size $LOCr(m_a) + mb_a$.</p> <p>This array contains the pivoting information.</p> <p>If $ipiv(i)=j$, then the local row <i>i</i> was swapped with the global row <i>j</i>.</p> <p>This array is tied to the distributed matrix <i>A</i>.</p>
<i>work</i> (1)	<p>On exit, <i>work</i>(1) contains the minimum value of <i>lwork</i> required for optimum performance.</p>
<i>iwork</i> (1)	<p>On exit, <i>iwork</i>(1) contains the minimum value of <i>liwork</i> required for optimum performance.</p>
<i>info</i>	<p>(global) INTEGER. If <i>info</i>=0, the execution is successful.</p> <p><i>info</i> < 0:</p> <p>If the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then $info = -(j*100+j)$; if the <i>i</i>-th argument is a scalar and had an illegal value, then $info = -i$.</p> <p><i>info</i> > 0:</p> <p>If $info = i$, $U(i,i)$ is exactly zero. The factorization has been completed, but the factor <i>U</i> is exactly singular, and division by zero will occur if it is used to solve a system of equations.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```

call pspotri(uplo, n, a, ia, ja, desca, info)
call pdpotri(uplo, n, a, ia, ja, desca, info)
call pcpotri(uplo, n, a, ia, ja, desca, info)
call pzpotri(uplo, n, a, ia, ja, desca, info)

```

Include Files

Description

The `p?potri` routine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ using the Cholesky factorization $\text{sub}(A) = U^H * U$ or $\text{sub}(A) = L * L^H$ computed by `p?potrf`.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If <code>uplo = 'U'</code> , upper triangle of $\text{sub}(A)$ is stored. If <code>uplo = 'L'</code> , lower triangle of $\text{sub}(A)$ is stored.
<code>n</code>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pspotri</code> DOUBLE PRECISION for <code>pdpotri</code> COMPLEX for <code>pcpotri</code> DOUBLE COMPLEX for <code>pzpotri</code> . Pointer into the local memory to an array of local size $(\text{lld}_a, \text{LOCc}(\text{ja} + n - 1))$. On entry, the array <code>a</code> contains the local pieces of the triangular factor U or L from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $\text{sub}(A) = L * L^H$, as computed by <code>p?potrf</code> .
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .

Output Parameters

<code>a</code>	On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of $\text{sub}(A)$.
<code>info</code>	(global) INTEGER. If <code>info=0</code> , the execution is successful. <code>info < 0</code> : If the i -th argument is an array and the j -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> . <code>info > 0</code> :

If $info = i$, the element (i, i) of the factor U or L is zero, and the inverse could not be computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
call pstrtri(uplo, diag, n, a, ia, ja, desca, info)
```

```
call pdtrtri(uplo, diag, n, a, ia, ja, desca, info)
```

```
call pctrtri(uplo, diag, n, a, ia, ja, desca, info)
```

```
call pztrtri(uplo, diag, n, a, ia, ja, desca, info)
```

Include Files

Description

The `p?trtri` routine computes the inverse of a real or complex upper or lower triangular distributed matrix $sub(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the distributed matrix $sub(A)$ is upper or lower triangular. If <i>uplo</i> = 'U', $sub(A)$ is upper triangular. If <i>uplo</i> = 'L', $sub(A)$ is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. Specifies whether or not the distributed matrix $sub(A)$ is unit triangular. If <i>diag</i> = 'N', then $sub(A)$ is non-unit triangular. If <i>diag</i> = 'U', then $sub(A)$ is unit triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix $sub(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for <code>pstrtri</code> DOUBLE PRECISION for <code>pdtrtri</code> COMPLEX for <code>pctrtri</code> DOUBLE COMPLEX for <code>pztrtri</code> . Pointer into the local memory to an array of local size $(lld_a, LOCC(ja+n-1))$. The array <i>a</i> contains the local pieces of the triangular distributed matrix $sub(A)$.

If `uplo = 'U'`, the leading n -by- n upper triangular part of `sub(A)` contains the upper triangular matrix to be inverted, and the strictly lower triangular part of `sub(A)` is not referenced.

If `uplo = 'L'`, the leading n -by- n lower triangular part of `sub(A)` contains the lower triangular matrix, and the strictly upper triangular part of `sub(A)` is not referenced.

`ia, ja`

(global) `INTEGER`. The row and column indices in the global matrix `A` indicating the first row and the first column of the matrix `sub(A)`, respectively.

`desca`

(global and local) `INTEGER` array of size `dlen_`. The array descriptor for the distributed matrix `A`.

Output Parameters

`a`

On exit, overwritten by the (triangular) inverse of the original matrix.

`info`

(global) `INTEGER`. If `info=0`, the execution is successful.

`info < 0`:

If the i -th argument is an array and the j -th entry had an illegal value, then `info = -(j*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

`info > 0`:

If `info = k`, `A(ia+k-1, ja+k-1)` is exactly zero. The triangular matrix `sub(A)` is singular and its inverse cannot be computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Routines for Matrix Equilibration

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
call psgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

```
call pdgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

```
call pcgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

```
call pzgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

Include Files

Description

The `p?geequ` routine computes row and column scalings intended to equilibrate an m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$ and reduce its condition number. The output array r returns the row scale factors r_i , and the array c returns the column scale factors c_j . These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r_i*a_{ij}*c_j$ have absolute value 1.

r_i and c_j are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of $\text{sub}(A)$ but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

The auxiliary function `p?laqge` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for <code>psgeequ</code> DOUBLE PRECISION for <code>pdgeequ</code> COMPLEX for <code>pcgeequ</code> DOUBLE COMPLEX for <code>pzgeequ</code> . Pointer into the local memory to an array of local size $(lld_a, LOCc(ja+n-1))$. The array a contains the local pieces of the m -by- n distributed matrix whose equilibration factors are to be computed.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

r, c	(local) REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays of sizes $LOCr(m_a)$ and $LOCc(n_a)$, respectively.
--------	---

If $info = 0$, or $info > ia+m-1$, the array $r(ia:ia+m-1)$ contains the row scale factors for $sub(A)$. r is aligned with the distributed matrix A , and replicated across every process column. r is tied to the distributed matrix A .

If $info = 0$, the array $c(ja:ja+n-1)$ contains the column scale factors for $sub(A)$. c is aligned with the distributed matrix A , and replicated down every process row. c is tied to the distributed matrix A .

rowcnd, colcnd

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If $info = 0$ or $info > ia+m-1$, *rowcnd* contains the ratio of the smallest $r(i)$ to the largest $r(i)$ ($ia \leq i \leq ia+m-1$). If $rowcnd \geq 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by $r(ia:ia+m-1)$.

If $info = 0$, *colcnd* contains the ratio of the smallest $c(j)$ to the largest $c(j)$ ($ja \leq j \leq ja+n-1$).

If $colcnd \geq 0.1$, it is not worth scaling by $c(ja:ja+n-1)$.

amax

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global) INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+i)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = i$ and

$i \leq m$, the i -th row of the distributed matrix

$sub(A)$ is exactly zero;

$i > m$, the $(i - m)$ -th column of the distributed

matrix $sub(A)$ is exactly zero.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
call pspoequ(n, a, ia, ja, desca, sr, sc, scnd, amax, info)
```

```
call pdpoequ(n, a, ia, ja, desca, sr, sc, scnd, amax, info)
```

```
call pcpcpoequ(n, a, ia, ja, desca, sr, sc, scnd, amax, info)
```

call pzpoequ(*n*, *a*, *ia*, *ja*, *desca*, *sr*, *sc*, *scond*, *amax*, *info*)

Include Files

Description

The pzpoequ routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ and reduce its condition number (with respect to the two-norm). The output arrays *sr* and *sc* return the row and column scale factors

, ! ! ! ! ! ! ! !

These factors are chosen so that the scaled distributed matrix *B* with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of *sr* and *sc* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function pzlaqsy uses scaling factors computed by pzggequ to scale a general rectangular matrix.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>a</i>	(local) REAL for pspoequ DOUBLE PRECISION for pdpoequ COMPLEX for pcpoequ DOUBLE COMPLEX for pzpoequ. Pointer into the local memory to an array of local size (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> - 1)). The array <i>a</i> contains the <i>n</i> -by- <i>n</i> symmetric/Hermitian positive definite distributed matrix sub(<i>A</i>) whose scaling factors are to be computed. Only the diagonal elements of sub(<i>A</i>) are referenced.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>sr</i> , <i>sc</i>	(local) REAL for single precision flavors;
-----------------------	---

DOUBLE PRECISION for double precision flavors.

Arrays of sizes $LOCr(m_a)$ and $LOCc(n_a)$, respectively.

If $info = 0$, the array $sr(ia:ia+n-1)$ contains the row scale factors for $sub(A)$. sr is aligned with the distributed matrix A , and replicated across every process column. sr is tied to the distributed matrix A .

If $info = 0$, the array $sc(ja:ja+n-1)$ contains the column scale factors for $sub(A)$. sc is aligned with the distributed matrix A , and replicated down every process row. sc is tied to the distributed matrix A .

scond

(global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If $info = 0$, *scond* contains the ratio of the smallest $sr(i)$ (or $sc(j)$) to the largest $sr(i)$ (or $sc(j)$), with

$ia \leq i \leq ia+n-1$ and $ja \leq j \leq ja+n-1$.

If $scond \geq 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by sr (or sc).

amax

(global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global) INTEGER.

If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = k$, the k -th diagonal entry of $sub(A)$ is nonpositive.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Orthogonal Factorizations

This section describes the ScaLAPACK routines for the $QR(RQ)$ and $LQ(QL)$ factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [[SLUG](#)].

[Table "Computational Routines for Orthogonal Factorizations"](#) lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqr p?unmqr
general matrices, RQ factorization	p?gerqf		p?orgrq p?ungrq	p?ormrq p?unmrq
general matrices, LQ factorization	p?gelqf		p?orglq p?unglq	p?ormlq p?unmlq
general matrices, QL factorization	p?geqlf		p?orgql p?ungql	p?ormql p?unmql
trapezoidal matrices, RZ factorization	p?tzzrf			p?ormrz p?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general m -by- n matrix.

Syntax

```
call psgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files**Description**

The p?geqrf routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia} + m - 1, \text{ja}:\text{ja} + n - 1)$ as

$$A = Q * R.$$

Input Parameters

m (global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$; ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$; ($n \geq 0$).

<i>a</i>	<p>(local)</p> <p>REAL for <code>psgeqrf</code></p> <p>DOUBLE PRECISION for <code>pdgeqrf</code></p> <p>COMPLEX for <code>pcgeqrf</code></p> <p>DOUBLE COMPLEX for <code>pzgeqrf</code>.</p> <p>Pointer into the local memory to an array of local size $(lld_a, LOCC(ja+n-1))$.</p> <p>Contains the local pieces of the distributed matrix $sub(A)$ to be factored.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A</p>
<i>work</i>	<p>(local).</p> <p>REAL for <code>psgeqrf</code></p> <p>DOUBLE PRECISION for <code>pdgeqrf</code>.</p> <p>COMPLEX for <code>pcgeqrf</code>.</p> <p>DOUBLE COMPLEX for <code>pzgeqrf</code></p> <p>Workspace array of size $lwork$.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of $work$, must be at least $lwork \geq nb_a * (mp0+nq0+nb_a)$, where</p> <p>$iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$,</p> <p>$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,</p> <p>$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,</p> <p>$mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW)$,</p> <p>$nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$, and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <code>blacs_gridinfo</code>.</p> <p>If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code>.</p>

Output Parameters

<i>a</i>	<p>The elements on and above the diagonal of $sub(A)$ contain the $\min(m,n)$-by-n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).</p>
<i>tau</i>	<p>(local)</p>

REAL for `psgeqrf`

DOUBLE PRECISION for `pdgeqrf`

COMPLEX for `pcgeqrf`

DOUBLE COMPLEX for `pzgeqrf`.

Array of size $LOCc(ja+\min(m,n)-1)$.

Contains the scalar factor of elementary reflectors. τ is tied to the distributed matrix A .

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

= 0, the execution is successful.

< 0, if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja)*H(ja+1)*...*H(ja+k-1),$$

where $k = \min(m,n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau*v*v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqpf

Computes the QR factorization of a general m -by- n matrix with pivoting.

Syntax

```
call psgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
```

```
call pdgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
```

```
call pcgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, rwork, lrwork, info)
```

```
call pzgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, rwork, lrwork, info)
```

Include Files

Description

The `p?geqpf` routine forms the QR factorization with column pivoting of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$\text{sub}(A)*P=Q*R.$$

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(A) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(A) ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>REAL for psgeqpf</p> <p>DOUBLE PRECISION for pdgeqpf</p> <p>COMPLEX for pcgeqpf</p> <p>DOUBLE COMPLEX for pzgeqpf.</p> <p>Pointer into the local memory to an array of local size (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i> + <i>n</i> - 1)).</p> <p>Contains the local pieces of the distributed matrix sub(A) to be factored.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A(<i>ia</i> : <i>ia</i> + <i>m</i> -1, <i>ja</i> : <i>ja</i> + <i>n</i> -1), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>work</i>	<p>(local).</p> <p>REAL for psgeqpf</p> <p>DOUBLE PRECISION for pdgeqpf.</p> <p>COMPLEX for pcgeqpf.</p> <p>DOUBLE COMPLEX for pzgeqpf</p> <p>Workspace array of size <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of <i>work</i>, must be at least</p> <p>For real flavors:</p> $lwork \geq \max(3, mp0 + nq0) + LOCc(ja + n - 1) + nq0.$ <p>For complex flavors:</p> $lwork \geq \max(3, mp0 + nq0).$ <p>Here</p> $iroff = \text{mod}(ia - 1, mb_a), \quad icoff = \text{mod}(ja - 1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mp0 = \text{numroc}(m + iroff, mb_a, MYROW, iarow, NPROW),$ $nq0 = \text{numroc}(n + icoff, nb_a, MYCOL, iacol, NPCOL),$ $LOCc(ja + n - 1) = \text{numroc}(ja + n - 1, nb_a, MYCOL, csrc_a, NPCOL),$ <p>and numroc, indxg2p are ScaLAPACK tool functions.</p> <p>You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfo subroutine.</p>

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

rwork

(local).

REAL for `pcgeqpf`.DOUBLE PRECISION for `pzgeqpf`.Workspace array of size $lrwork$ (complex flavors only).*lrwork*(local or global) INTEGER, size of *rwork* (complex flavors only). The value of $lrwork$ must be at least $lwork \geq LOCC(ja+n-1) + nq0$.

Here

 $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW)$, $nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$, $LOCC(ja+n-1) = \text{numroc}(ja+n-1, nb_a, MYCOL, csrc_a, NPCOL)$,and `numroc`, `indxg2p` are ScaLAPACK tool functions.You can determine `MYROW`, `MYCOL`, `NPROW` and `NPCOL` by calling the `blacs_gridinfo` subroutine.

If $lrwork = -1$, then $lrwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

The elements on and above the diagonal of `sub(A)` contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

ipiv(local) INTEGER. Array of size $LOCC(ja+n-1)$.

$ipiv(i) = k$, the local i -th column of `sub(A)*P` was the global k -th column of `sub(A)`. *ipiv* is tied to the distributed matrix A .

tau

(local)

REAL for `psgeqpf`DOUBLE PRECISION for `pdgeqpf`COMPLEX for `pcgeqpf`DOUBLE COMPLEX for `pzgeqpf`.

	Array of size $LOCc(ja+\min(m, n)-1)$.
	Contains the scalar factor τ of elementary reflectors. τ is tied to the distributed matrix A .
$work(1)$	On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$rwork(1)$	On exit, $rwork(1)$ contains the minimum value of $lrwork$ required for optimum performance.
$info$	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1)*H(2)*...*H(k)$$

where $k = \min(m,n)$.

Each $H(i)$ has the form

$$H = I - \tau*v*v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$.

The matrix P is represented in $ipiv$ as follows: if $ipiv(j) = i$ then the j -th column of P is the i -th canonical unit vector.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The p?orgqrroutine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1)*H(2)*...*H(k)$$

as returned by p?geqrf.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(<i>Q</i>) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(<i>Q</i>) ($m \geq n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($n \geq k \geq 0$).
<i>a</i>	(local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Pointer into the local memory to an array of local size (<i>lld_a</i> , <i>LOCc</i> (<i>ja</i> + <i>n</i> - 1)). The <i>j</i> -th column must contain the vector that defines the elementary reflector <i>H</i> (<i>j</i>), $ja \leq j \leq ja + k - 1$, as returned by p?geqrf in the <i>k</i> columns of its distributed matrix argument <i>A</i> (<i>ia</i> : *, <i>ja</i> : <i>ja</i> + <i>k</i> - 1).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> (<i>ia</i> : <i>ia</i> + <i>m</i> - 1, <i>ja</i> : <i>ja</i> + <i>n</i> - 1), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Array of size <i>LOCc</i> (<i>ja</i> + <i>k</i> - 1). Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors <i>H</i> (<i>j</i>) as returned by p?geqrf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psorgqr DOUBLE PRECISION for pdorgqr Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> . Must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where <i>irowfa</i> = mod(<i>ia</i> - 1, <i>mb_a</i>), <i>icoffa</i> = mod(<i>ja</i> - 1, <i>nb_a</i>), <i>iarow</i> = indxg2p(<i>ia</i> , <i>mb_a</i> , MYROW, <i>rsrc_a</i> , NPROW), <i>iacol</i> = indxg2p(<i>ja</i> , <i>nb_a</i> , MYCOL, <i>csrc_a</i> , NPCOL), <i>mpa0</i> = numroc(<i>m</i> + <i>irowfa</i> , <i>mb_a</i> , MYROW, <i>iarow</i> , NPROW), <i>nqa0</i> = numroc(<i>n</i> + <i>icoffa</i> , <i>nb_a</i> , MYCOL, <i>iacol</i> , NPCOL); <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work(1)</code>	On exit, (1) contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+i)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
call pcungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1)*H(2)*...*H(k)$$

as returned by `p?geqrf`.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the matrix sub(Q); ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the matrix sub(Q) ($m \geq n \geq 0$).
<code>k</code>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<code>a</code>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code>

Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$. The j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by `p?geqrf` in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code> Array of size $LOCc(ja+k-1)$. Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by <code>p?geqrf</code> . τ is tied to the distributed matrix A .
<i>work</i>	(local) COMPLEX for <code>pcungqr</code> DOUBLE COMPLEX for <code>pzungqr</code> Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW)$, $nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <code>blacs_gridinfo</code> . If $lwork = -1$, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix Q .
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call psormqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

The p?ormqr routine overwrites the general real m -by- n distributed matrix sub (C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'T':$	$Q^T * sub(C)$	$sub(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by p?geqrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER = 'L' : Q or Q^T is applied from the left. = 'R' : Q or Q^T is applied from the right.
$trans$	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).

<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$. The <i>j</i> -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$ If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array of size $LOCc(ja+k-1)$. Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by p?geqrf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the matrix sub(<i>C</i>), respectively.
<i>desc</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

if *side* = 'L',

$$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$$

else if *side* = 'R',

$$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$$

end if

where

$$lcmq = lcm / NPCOL \text{ with } lcm = ilcm(NPROW, NPCOL),$$

$$iroffa = \text{mod}(ia - 1, mb_a),$$

$$icoffa = \text{mod}(ja - 1, nb_a),$$

$$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$$

$$npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW),$$

$$iroffc = \text{mod}(ic - 1, mb_c),$$

$$icoffc = \text{mod}(jc - 1, nb_c),$$

$$icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW),$$

$$iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL),$$

$$mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW),$$

$$nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL),$$

ilcm, *indxg2p* and *numroc* are **ScaLAPACK** tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *p_xerbla*.

Output Parameters

c

Overwritten by the product $Q * \text{sub}(C)$, or $Q^T * \text{sub}(C)$, or $\text{sub}(C) * Q^T$, or $\text{sub}(C) * Q$.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

```
call pcunmqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k)$ as returned by p?geqrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER = $'L'$: Q or Q^H is applied from the left. = $'R'$: Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = $'N'$, no transpose, Q is applied. = $'C'$, conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmqr

	<p>DOUBLE COMPLEX for pzunmqr.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+k-1))$. The j-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.</p> <p>If $side = 'L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$</p> <p>If $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	<p>(local)</p> <p>COMPLEX for pcunmqr</p> <p>DOUBLE COMPLEX for pzunmqr</p> <p>Array of size $LOCc(ja+k-1)$.</p> <p>Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by <code>p?geqrf</code>. τ is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>COMPLEX for pcunmqr</p> <p>DOUBLE COMPLEX for pzunmqr.</p> <p>Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$.</p> <p>Contains the local pieces of the distributed matrix sub(C) to be factored.</p>
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.
<i>descc</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .
<i>work</i>	<p>(local)</p> <p>COMPLEX for pcunmqr</p> <p>DOUBLE COMPLEX for pzunmqr.</p> <p>Workspace array of size of $lwork$.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of $work$, must be at least:</p> <p>If $side = 'L'$,</p> <p>$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$</p> <p>else if $side = 'R'$,</p>

```

lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0 +
numroc(numroc(n+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0,
lcmq), mpc0))*nb_a) + nb_a*nb_a

```

end if

where

```

lcmq = lcm/NPCOL with lcm = ilcm (NPROW, NPCOL),
irowfa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(n+irowfa, mb_a, MYROW, iarow, NPROW),
irowfc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+irowfc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the subroutine
blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p_xerbla`.

Output Parameters

<i>c</i>	Overwritten by the product $Q*\text{sub}(C)$, or $Q^H*\text{sub}(C)$, or $\text{sub}(C)*Q^H$, or $\text{sub}(C)*Q$.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>j</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```
call psgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?gelqf` routine computes the LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for <code>psgelqf</code> DOUBLE PRECISION for <code>pdgelqf</code> COMPLEX for <code>pcgelqf</code> DOUBLE COMPLEX for <code>pzgelqf</code> Pointer into the local memory to an array of local size $(l1d_a, LOCC(ja+n-1))$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local) REAL for <code>psgelqf</code> DOUBLE PRECISION for <code>pdgelqf</code> COMPLEX for <code>pcgelqf</code> DOUBLE COMPLEX for <code>pzgelqf</code> Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iroff = \text{mod}(ia-1, mb_a)$,

```

icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL)

```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

NOTE

`mod(x,y)` is the integer remainder of x/y .

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	The elements on and below the diagonal of sub(A) contain the m -by- $\min(m,n)$ lower trapezoidal matrix L (L is lower trapezoidal if $m \leq n$); the elements above the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Array of size $LOCr(ia+\min(m, n)-1)$. Contains the scalar factors of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <code>lwork</code> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+i)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia+k-1)*H(ia+k-2)*...*H(ia),$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and τ in $\tau(ia+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by [p?gelqf](#).

Syntax

```
call psorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?orglq` routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by [p?gelqf](#).

Input Parameters

m	(global) INTEGER. The number of rows in the matrix sub(Q); ($m \geq 0$).
n	(global) INTEGER. The number of columns in the matrix sub(Q) ($n \geq m \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
a	(local) REAL for <code>psorglq</code> DOUBLE PRECISION for <code>pdorglq</code> Pointer into the local memory to an array of local size $(lld_a, LOCC(ja+n-1))$. On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

work (local)
 REAL for psorglq
 DOUBLE PRECISION for pdorglq
 Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least
 $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where

```

irow = mod(ia-1, mb_a),
icol = mod(ja-1, nb_a),
mpa0 = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
nqa0 = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+irow, mb_a, MYROW, icol, NPROW),
nqa0 = numroc(n+icol, nb_a, MYCOL, irow, NPCOL)

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a Contains the local pieces of the m -by- n distributed matrix Q to be factored.

tau (local)
 REAL for psorglq
 DOUBLE PRECISION for pdorglq
 Array of size $LOCr(ia+k-1)$.
 Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. τ is tied to the distributed matrix A .

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+i)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by `p?gelqf`.

Syntax

```
call pcunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pzunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by `p?gelqf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(Q) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(Q) ($n \geq m \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Pointer into the local memory to an array of local size (<code>lld_a, LOcc(ja+n-1)</code>). On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Array of size <code>LOCr(ia+k-1)</code> . Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. τ is tied to the distributed matrix A .
<i>work</i>	(local) COMPLEX for <code>pcunglq</code>

DOUBLE COMPLEX for pzunglq

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least

$lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where

$irowfa = \text{mod}(ia-1, mb_a)$,

$icoffa = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,

$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,

$mpa0 = \text{numroc}(m+irowfa, mb_a, MYROW, iarow, NPROW)$,

$nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$

`indxg2p` and `numroc` are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine `blacs_gridinfo`.

NOTE

`mod(x, y)` is the integer remainder of x/y .

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

Contains the local pieces of the m -by- n distributed matrix Q to be factored.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

call psormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

call pdormlq(*side*, *trans*, *m*, *n*, *k*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *work*, *lwork*, *info*)

Include Files

Description

The `p?ormlq` routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<i>trans</i> = 'T':	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2) H(1)$$

as returned by `p?gelqf`. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for <code>psormlq</code> DOUBLE PRECISION for <code>pdormlq</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+m-1))$, if <i>side</i> = 'L' and $(lld_a, LOCC(ja+n-1))$, if <i>side</i> = 'R'. The i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)
 REAL for psormlq
 DOUBLE PRECISION for pdormlq
 Array of size $LOCc(ja+k-1)$.
 Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.

c (local)
 REAL for psormlq
 DOUBLE PRECISION for pdormlq
 Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$.
 Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, jc (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

desc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)
 REAL for psormlq
 DOUBLE PRECISION for pdormlq.
 Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of the array *work*; must be at least:
 If *side* = 'L',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + maxmqa0) + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a + mb_a * mb_a$
 else if *side* = 'R',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a + mb_a * mb_a$
 end if
 where
 $lcmp = lcm / NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(ia - 1, mb_a)$,
 $icoffa = \text{mod}(ja - 1, nb_a)$,
 $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,
 $mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL)$,
 $iroffc = \text{mod}(ic - 1, mb_c)$,

```

icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x, y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?erbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
call pcunmlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q*\text{sub}(C)$	$\text{sub}(C)*Q$
<i>trans</i> = 'T':	$Q^H*\text{sub}(C)$	$\text{sub}(C)*Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by `p?gelsqf`. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix <code>sub(C)</code> ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix <code>sub(C)</code> ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for <code>pcunmlq</code> DOUBLE COMPLEX for <code>pzunmlq</code> . Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$, if <i>side</i> = 'L' and $(lld_a, LOCc(ja+n-1))$, if <i>side</i> = 'R', where $lld_a \geq \max(1, LOCr(ia+k-1))$. The i -th column must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelsqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for <code>pcunmlq</code> DOUBLE COMPLEX for <code>pzunmlq</code>

	<p>Array of size $LOCc(ia+k-1)$.</p> <p>Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by <code>p?gelqf</code>. τ is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>COMPLEX for <code>pcunmlq</code></p> <p>DOUBLE COMPLEX for <code>pzunmlq</code>.</p> <p>Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$.</p> <p>Contains the local pieces of the distributed matrix $\text{sub}(C)$ to be factored.</p>
<i>ic, jc</i>	<p>(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.</p>
<i>desc</i>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C.</p>
<i>work</i>	<p>(local)</p> <p>COMPLEX for <code>pcunmlq</code></p> <p>DOUBLE COMPLEX for <code>pzunmlq</code>.</p> <p>Workspace array of size of $lwork$.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of the array $work$; must be at least:</p> <p>If $side = 'L'$,</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a)$ <p>else if $side = 'R'$,</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a + mb_a * mb_a)$ <p>end if</p> <p>where</p> $lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL),$ $iroffc = \text{mod}(ic - 1, mb_c),$ $icoffc = \text{mod}(jc - 1, nb_c),$ $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW),$ $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL),$ $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW),$ $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL),$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pserbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(j \cdot 100 + i)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```
call psgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files**Description**

The `p?geqlf` routine forms the QL factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(\text{ia}:\text{ia} + m - 1, \text{ja}:\text{ja} + n - 1) = Q \cdot L$.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the matrix $\text{sub}(Q)$; ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the matrix $\text{sub}(Q)$ ($n \geq 0$).

<i>a</i>	<p>(local)</p> <p>REAL for psgeqlf</p> <p>DOUBLE PRECISION for pdgeqlf</p> <p>COMPLEX for pcgeqlf</p> <p>DOUBLE COMPLEX for pzgeqlf</p> <p>Pointer into the local memory to an array of local size $(lld_a, LOCC(ja + n - 1))$. Contains the local pieces of the distributed matrix sub(<i>A</i>) to be factored.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>work</i>	<p>(local)</p> <p>REAL for psgeqlf</p> <p>DOUBLE PRECISION for pdgeqlf</p> <p>COMPLEX for pcgeqlf</p> <p>DOUBLE COMPLEX for pzgeqlf</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of <i>work</i>, must be at least $lwork \geq nb_a * (mp0 + nq0 + nb_a)$, where</p> <p>$iroff = \text{mod}(ia-1, mb_a)$,</p> <p>$icoff = \text{mod}(ja-1, nb_a)$,</p> <p>$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,</p> <p>$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,</p> <p>$mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW)$,</p> <p>$nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$</p>

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`numroc` and `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) REAL for psgeqlf DOUBLE PRECISION for pdgeqlf COMPLEX for pcgeqlf DOUBLE COMPLEX for pzgeqlf Array of size $LOCc(ja+n-1)$. Contains the scalar factors of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) \dots H(ja+1) H(ja)$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $\tau(ja+n-k+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call psorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?orgql` routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by `p?geqlf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(Q), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(Q), ($m \geq n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Pointer into the local memory to an array of local size (<code>lld_a, LOCC(ja+n-1)</code>). On entry, the j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by <code>p?geqlf</code> in the k columns of its distributed matrix argument $A(ia:*, ja+n-k:ja+n-1)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Array of size <code>LOCC(ja+n-1)</code> . Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. τ is tied to the distributed matrix A .
<i>work</i>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Workspace array of size of <code>lwork</code> .
<i>lwork</i>	(local or global) INTEGER, size of <code>work</code> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a)$,

```

icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungql

Generates the unitary matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```

call pcungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

```

Include Files**Description**

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first n columns of a product of k elementary reflectors of order m

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by `p?geqlf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(<i>Q</i>) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(<i>Q</i>) ($m \geq n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($n \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Pointer into the local memory to an array of local size (<i>lld_a</i> , <i>LOCc</i> (<i>ja+n-1</i>)). On entry, the <i>j</i> -th column must contain the vector that defines the elementary reflector <i>H</i> (<i>j</i>), $ja+n-k \leq j \leq ja+n-1$, as returned by p?geqlf in the <i>k</i> columns of its distributed matrix argument <i>A</i> (<i>ia</i> :* , <i>ja+n-k</i> : <i>ja+n-1</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> (<i>ia</i> : <i>ia+m-1</i> , <i>ja</i> : <i>ja+n-1</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Array of size <i>LOCr</i> (<i>ia+n-1</i>). Contains the scalar factors <i>tau</i> (<i>j</i>) of elementary reflectors <i>H</i> (<i>j</i>). <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $irow = \text{mod}(ia-1, mb_a)$, $icol = \text{mod}(ja-1, nb_a)$, $irow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $icol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mpa0 = \text{numroc}(m+irow, mb_a, MYROW, irow, NPROW)$, $nqa0 = \text{numroc}(n+icol, nb_a, MYCOL, icol, NPCOL)$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
call psormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

The `p?ormql` routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<code>side</code>	(global) CHARACTER = 'L': Q or Q^T is applied from the left.
-------------------	---

	= 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C), ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormql DOUBLE PRECISION for pdormql. Pointer into the local memory to an array of size ($lld_a, LOCc(ja+k-1)$). The j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?ge1qf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for psormql DOUBLE PRECISION for pdormql. Array of size $LOCc(ja+n-1)$. Contains the scalar factor $tau(j)$ of elementary reflectors $H(j)$ as returned by p?geqlf. tau is tied to the distributed matrix A .
<i>c</i>	(local) REAL for psormql DOUBLE PRECISION for pdormql. Pointer into the local memory to an array of local size ($lld_c, LOCc(jc+n-1)$).

	Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>desc</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormql DOUBLE PRECISION for pdormql. Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a + nb_a * nb_a)$ else if <i>side</i> = 'R', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + iroffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a + nb_a * nb_a)$ end if where $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$, $iroffa = \text{mod}(ia - 1, mb_a)$, $icoffa = \text{mod}(ja - 1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW)$, $iroffc = \text{mod}(ic - 1, mb_c)$, $icoffc = \text{mod}(jc - 1, nb_c)$, $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$, $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$, $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$, $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+i)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```
call pcunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
            lwork, info)
```

```
call pzunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
            lwork, info)
```

Include Files

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q^* \text{sub}(C)$	$\text{sub}(C)^* Q$
$trans = 'C':$	$Q^H \text{sub}(C)$	$\text{sub}(C)^H Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<code>side</code>	(global) CHARACTER = 'L': Q or Q^H is applied from the left.
-------------------	---

	= 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+k-1))$. The j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqlf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql Array of size $LOCC(ia+n-1)$. Contains the scalar factor $\tau(j)$ of elementary reflectors $H(j)$ as returned by <code>p?geqlf</code> . τ is tied to the distributed matrix A .
<i>c</i>	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql. Pointer into the local memory to an array of local size $(lld_c, LOCC(jc+n-1))$.

	Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>desc</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql. Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a + nb_a * nb_a)$ else if <i>side</i> = 'R', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0) + \text{numroc}(\text{numroc}(n + ioffset, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a + nb_a * nb_a)$ end if where $lcmp = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$, $iroffa = \text{mod}(ia - 1, mb_a)$, $icoffa = \text{mod}(ja - 1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW)$, $iroffc = \text{mod}(ic - 1, mb_c)$, $icoffc = \text{mod}(jc - 1, nb_c)$, $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$, $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$, $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$, $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q' \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of $lwork$ required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```
call psggerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files**Description**

The `p?gerqf` routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = R^*Q$$

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$; ($m \geq 0$).
----------------	--

<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(A); ($n \geq 0$).
<i>a</i>	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Pointer into the local memory to an array of local size ($lld_a, LOCC(ja+n-1)$). Contains the local pieces of the distributed matrix sub(A) to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A
<i>work</i>	(local). REAL for psgeqrf DOUBLE PRECISION for pdgeqrf. COMPLEX for pcgeqrf. DOUBLE COMPLEX for pzgeqrf Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

$nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$ and numroc , indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m - n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Array of size $LOCr(ia+m-1)$. Contains the scalar factor of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ in $\tau(ia+m-k+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgrq

Generates the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psorgrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdorgrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?orgrq` routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows that is defined as the last m rows of a product of k elementary reflectors of order n

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `p?gerqf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(Q), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(Q), ($n \geq m \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) REAL for <code>psorgrq</code> DOUBLE PRECISION for <code>pdorgrq</code> Pointer into the local memory to an array of local size (<code>lld_a, LOcc(ja+n-1)</code>). The i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+m-1$, as returned by <code>p?gerqf</code> in the k rows of its distributed matrix argument $A(ia+m-k:ia+m-1, ja:*)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for <code>psorgrq</code> DOUBLE PRECISION for <code>pdorgrq</code> Array of size <code>LOcc(ja+k-1)</code> . Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by <code>p?gerqf</code> . τ is tied to the distributed matrix A .
<i>work</i>	(local) REAL for <code>psorgrq</code> DOUBLE PRECISION for <code>pdorgrq</code> Workspace array of size of <code>lwork</code> .
<i>lwork</i>	(local or global) INTEGER, size of <code>work</code> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a)$,

```

icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix Q .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of $lwork$ required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```

call pcungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

```

Include Files**Description**

This routine generates the m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by [p?gerqf](#).

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the matrix sub(<i>Q</i>); ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the matrix sub(<i>Q</i>) ($n \geq m \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> ($m \geq k \geq 0$).
<i>a</i>	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrq Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$. The <i>i</i> -th row must contain the vector that defines the elementary reflector <i>H</i> (<i>i</i>), $ia+m-k \leq i \leq ia+m-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument <i>A</i> ($ia+m-k:ia+m-1, ja:*$).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrq Array of size $LOCr(ia+m-1)$. Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors <i>H</i> (<i>i</i>) as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrq Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $irow = \text{mod}(ia-1, mb_a)$, $icol = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mpa0 = \text{numroc}(m+irow, mb_a, MYROW, iarow, NPROW)$, $nqa0 = \text{numroc}(n+icol, nb_a, MYCOL, icol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(j*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
call psormrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files**Description**

The `p?ormrq` routine overwrites the general real m -by- n distributed matrix sub (C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q*\text{sub}(C)$	$\text{sub}(C)*Q$
<code>trans = 'T':</code>	$Q^T*\text{sub}(C)$	$\text{sub}(C)*Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by `p?gerqf`. Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCC(ja+n-1))$ if <i>side</i> = 'R'. The <i>i</i> -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq j \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array of size $LOCC(ja+k-1)$. Contains the scalar factor $tau(i)$ of elementary reflectors $H(i)$ as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix A .
<i>c</i>	(local) REAL for psormqr

DOUBLE PRECISION for `pdormrq`

Pointer into the local memory to an array of local size $(lld_c, LOCc(jc + n - 1))$.

Contains the local pieces of the distributed matrix $sub(C)$ to be factored.

ic, jc (global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the matrix $sub(C)$, respectively.

desc (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work (local)
 REAL for `psormrq`
 DOUBLE PRECISION for `pdormrq`.
 Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:
 If *side* = 'L',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a)$
 else if *side* = 'R',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$
 end if

where

$lcmp = lcm / NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(ia - 1, mb_a)$,
 $icoffa = \text{mod}(ja - 1, nb_a)$,
 $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,
 $mqa0 = \text{numroc}(n + icoffa, nb_a, MYCOL, iacol, NPCOL)$,
 $iroffc = \text{mod}(ic - 1, mb_c)$,
 $icoffc = \text{mod}(jc - 1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$,
 $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$,
 $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$,
 $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q * \text{sub}(C)$, or $Q' * \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by `p?gerqf`.

Syntax

```
call pcunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<code>trans = 'C':</code>	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by `p?gerqf`. Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C), ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCC(ja+n-1))$ if <i>side</i> = 'R'. The i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq j \leq ia$ $+k-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq Array of size $LOCC(ja+k-1)$. Contains the scalar factor $tau(i)$ of elementary reflectors $H(i)$ as returned by p?gerqf. tau is tied to the distributed matrix A .
<i>c</i>	(local) COMPLEX for pcunmrq DOUBLE COMPLEX for pzunmrq.

	<p>Pointer into the local memory to an array of local size $(lld_c, LOcc(jc + n - 1))$.</p> <p>Contains the local pieces of the distributed matrix sub(C) to be factored.</p>
<i>ic, jc</i>	<p>(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.</p>
<i>desc</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix C.</p>
<i>work</i>	<p>(local)</p> <p>COMPLEX for pcunmrq</p> <p>DOUBLE COMPLEX for pzunmrq.</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) INTEGER, size of <i>work</i>, must be at least:</p> <p>If <i>side</i> = 'L',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a$ <p>else if <i>side</i> = 'R',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$ <p>end if</p> <p>where</p> $lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL),$ $iroffc = \text{mod}(ic - 1, mb_c),$ $icoffc = \text{mod}(jc - 1, nb_c),$ $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW),$ $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL),$ $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW),$ $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL),$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c	Overwritten by the product $Q^* \text{sub}(C)$ or $Q^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
call pxtzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdtzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pctzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pstzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?tzrzf` routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$$A = (R \ 0)^* Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows in the matrix $\text{sub}(A)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for <code>p_stzrzf</code>

DOUBLE PRECISION for `pdtzrzf`.

COMPLEX for `pctzrzf`.

DOUBLE COMPLEX for `pztzrzf`.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. Contains the local pieces of the m -by- n distributed matrix sub (A) to be factored.

ia, ja

(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A .

work

(local)

REAL for `pstzrzf`

DOUBLE PRECISION for `pdtzrzf`.

COMPLEX for `pctzrzf`.

DOUBLE COMPLEX for `pztzrzf`.

Workspace array of size of *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least

$lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where

$irow = \text{mod}(ia-1, mb_a)$,

$icoff = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,

$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,

$mp0 = \text{numroc}(m+irow, mb_a, MYROW, iarow, NPROW)$,

$nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

τ is a scalar and $Z(k)$ is an $(n - m)$ element vector. τ and $Z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $Z(k)$ are in $a(k, m + 1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$. Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzzrf.

Syntax

```
call psormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by p?tzzrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER $= 'L'$: Q or Q^T is applied from the left. $= 'R'$: Q or Q^T is applied from the right.
$trans$	(global) CHARACTER $= 'N'$, no transpose, Q is applied. $= 'T'$, transpose, Q^T is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).

<i>k</i>	<p>(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i>. Constraints:</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>l</i>	<p>(global)</p> <p>The columns of the distributed matrix sub(<i>A</i>) containing the meaningful part of the Householder reflectors.</p> <p>If <i>side</i> = 'L', $m \geq l \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq l \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>REAL for psormrz</p> <p>DOUBLE PRECISION for pdormrz.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCc(ja+n-1))$ if <i>side</i> = 'R', where $lld_a \geq \max(1, LOCr(ia+k-1))$.</p> <p>The <i>i</i>-th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?tzrzf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>REAL for psormrz</p> <p>DOUBLE PRECISION for pdormrz</p> <p>Array of size $LOCc(ia+k-1)$.</p> <p>Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by p?tzrzf. <i>tau</i> is tied to the distributed matrix <i>A</i>.</p>
<i>c</i>	<p>(local)</p> <p>REAL for psormrz</p> <p>DOUBLE PRECISION for pdormrz</p> <p>Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$.</p> <p>Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.</p>
<i>ic, jc</i>	<p>(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i>, respectively.</p>

desc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)
 REAL for *psormrz*
 DOUBLE PRECISION for *pdormrz*.
 Workspace array of size of *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:
 If *side* = 'L',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a)$
 else if *side* = 'R',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$
 end if
 where
 $lcmp = lcm / NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(ia - 1, mb_a)$, $icoffa = \text{mod}(ja - 1, nb_a)$,
 $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,
 $mqa0 = \text{numroc}(n + icoffa, nb_a, MYCOL, iacol, NPCOL)$,
 $iroffc = \text{mod}(ic - 1, mb_c)$,
 $icoffc = \text{mod}(jc - 1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$,
 $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$,
 $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$,
 $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

c Overwritten by the product $Q * \text{sub}(C)$, or $Q' * \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$

`work(1)` On exit `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info` (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -th entry had an illegal value, then `info` = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzzrf.

Syntax

```
call pcunmrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general complex m -by- n distributed matrix sub (C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'C':$	$Q^H * sub(C)$	$sub(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by `pctzrzf/pztzrzf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

`side` (global) CHARACTER
 = 'L': Q or Q^H is applied from the left.
 = 'R': Q or Q^H is applied from the right.

`trans` (global) CHARACTER
 = 'N', no transpose, Q is applied.
 = 'C', conjugate transpose, Q^H is applied.

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C), ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>l</i>	(global) INTEGER. The columns of the distributed matrix sub(A) containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$ If <i>side</i> = 'R', $n \geq l \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCc(ja+n-1))$ if <i>side</i> = 'R', where $lld_a \geq \max(1, LOCr(ja+k-1))$. The <i>i</i> -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>tau</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz Array of size $LOCc(ia+k-1)$. Contains the scalar factor $\tau(i)$ of elementary reflectors $H(i)$ as returned by p?gerqf . <i>tau</i> is tied to the distributed matrix A.
<i>c</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of local size $(lld_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub(C) to be factored.

ic, jc (global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

desc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local)
 COMPLEX for *pcunmrz*
 DOUBLE COMPLEX for *pzunmrz*.
 Workspace array of size *lwork*.

lwork (local or global) INTEGER, size of *work*, must be at least:
 If *side* = 'L',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a + mb_a * mb_a)$
 else if *side* = 'R',
 $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$
 end if
 where
 $lcmp = lcm / NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,
 $iroffa = \text{mod}(ia - 1, mb_a)$,
 $icoffa = \text{mod}(ja - 1, nb_a)$,
 $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,
 $mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL)$,
 $iroffc = \text{mod}(ic - 1, mb_c)$,
 $icoffc = \text{mod}(jc - 1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$,
 $iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$,
 $mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$,
 $nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $\text{info} = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ggqrf

Computes the generalized QR factorization.

Syntax

```
call psggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pdggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pcggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pzggqrf(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

Include Files

Description

The `p?ggqrf` routine forms the generalized QR factorization of an n -by- m matrix

$$\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$$

and an n -by- p matrix

$$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+p-1):$$

as

$$\text{sub}(A) = Q^* R, \text{sub}(B) = Q^* T^* Z,$$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

If $n \geq m$

	<p>Pointer into the local memory to an array of size $(lld_a, LOCC(ja+m-1))$. Contains the local pieces of the n-by-m matrix $sub(A)$ to be factored.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A.</p>
<i>b</i>	<p>(local)</p> <p>REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf.</p> <p>Pointer into the local memory to an array of size $(lld_b, LOCC(jb+p-1))$. Contains the local pieces of the n-by-p matrix $sub(B)$ to be factored.</p>
<i>ib, jb</i>	<p>(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B, respectively.</p>
<i>descb</i>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B.</p>
<i>work</i>	<p>(local)</p> <p>REAL for psggqrf DOUBLE PRECISION for pdggqrf COMPLEX for pcggqrf DOUBLE COMPLEX for pzggqrf.</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) INTEGER. Size of <i>work</i>, must be at least</p> $lwork \geq \max(nb_a * (npa0 + mqa0 + nb_a), \max((nb_a * (nb_a - 1)) / 2, (pqb0 + npb0) * nb_a + nb_a * nb_a, mb_b * (npb0 + pqb0 + mb_b)),$ <p>where</p> $iroffa = \text{mod}(ia - 1, mb_A),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW),$ $mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL)$ $iroffb = \text{mod}(ib - 1, mb_b),$ $icoffb = \text{mod}(jb - 1, nb_b),$ $ibrow = \text{indxg2p}(ib, mb_b, MYROW, rsrc_b, NPROW),$

```

ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
npb0 = numroc (n+iroffa, mb_b, MYROW, Ibrow, NPROW),
pqb0 = numroc(m+icoffb, nb_b, MYCOL, ibcol, NPCOL)

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

and `numroc`, `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters`a`

On exit, the elements on and above the diagonal of sub (*A*) contain the $\min(n, m)$ -by- m upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$); the elements below the diagonal, with the array `taua`, represent the orthogonal/unitary matrix *Q* as a product of $\min(n, m)$ elementary reflectors. (See Application Notes below).

`taua, taub`

(local)

REAL for `psggqrf`DOUBLE PRECISION for `pdggqrf`COMPLEX for `pcggqrf`DOUBLE COMPLEX for `pzggqrf`.

Arrays of size `LOC(ja+min(n,m)-1)` for `taua` and `LOC(ib+n-1)` for `taub`.

The array `taua` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. `taua` is tied to the distributed matrix *A*. (See Application Notes below).

The array `taub` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. `taub` is tied to the distributed matrix *B*. (See Application Notes below).

`work(1)`

On exit `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info = -(i*100+j)`; if the *i*-th argument is a scalar and had an illegal value, then `info = -i`.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(j_a) * H(j_a+1) * \dots * H(j_a+k-1),$$

where $k = \min(n, m)$.

Each $H(i)$ has the form

$$H(i) = I - \tau_a * v * v'$$

where τ_a is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1, ja+i-1)$, and τ_a in $\tau_a(ja+i-1)$. To form Q explicitly, use ScaLAPACK subroutine `p?orgqr/p?ungqr`. To use Q to update another matrix, use ScaLAPACK subroutine `p?ormqr/p?unmqr`.

The matrix Z is represented as a product of elementary reflectors

$$Z = H(ib) * H(ib+1) * \dots * H(ib+k-1), \text{ where } k = \min(n, p).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_b * v * v'$$

where τ_b is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1, jb:jb+p-k+i-2)$, and τ_b in $\tau_b(ib+n-k+i-1)$. To form Z explicitly, use ScaLAPACK subroutine `p?orgqr/p?ungqr`. To use Z to update another matrix, use ScaLAPACK subroutine `p?ormr/p?unmr`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

`p?ggrqf`

Computes the generalized RQ factorization.

Syntax

```
call psggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pdggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pcggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

```
call pzggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

Include Files

Description

The `p?ggrqf` routine forms the generalized RQ factorization of an m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ and a p -by- n matrix $\text{sub}(B) = B(ib:ib+p-1, jb:jb+n-1)$:

$$\text{sub}(A) = R * Q, \text{ sub}(B) = Z * T * Q,$$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} m & (0 & R_{12}) \\ & n - m & m \end{matrix}, \text{ if } m \leq n,$$

or

$$R = \begin{matrix} \begin{pmatrix} R_{11} \\ R_{12} \end{pmatrix} & m - n \\ & n \end{matrix}, \text{ if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} & n \\ & p - n \end{matrix}, \text{ if } p \geq n$$

or

$$\begin{matrix} \dots\dots\dots & \dots\dots\dots \\ \dots\dots\dots & \dots\dots\dots \end{matrix}$$

where T_{11} is upper triangular.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GRQ factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the RQ factorization of $\text{sub}(A) \cdot \text{inv}(\text{sub}(B))$:

$$\text{sub}(A) \cdot \text{inv}(\text{sub}(B)) = (R \cdot \text{inv}(T)) \cdot Z'$$

where $\text{inv}(\text{sub}(B))$ denotes the inverse of the matrix $\text{sub}(B)$, and Z' denotes the transpose (conjugate transpose) of matrix Z .

Input Parameters

- m (global) INTEGER. The number of rows in the distributed matrices $\text{sub}(A)$ ($m \geq 0$).
- p INTEGER. The number of rows in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).
- n (global) INTEGER. The number of columns in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
- a (local)
 REAL for psggrqf
 DOUBLE PRECISION for pdggrqf

	<p>COMPLEX for <code>pcggrqf</code></p> <p>DOUBLE COMPLEX for <code>pzggrqf</code>.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$. Contains the local pieces of the m-by-n distributed matrix $\text{sub}(A)$ to be factored.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>b</i>	<p>(local)</p> <p>REAL for <code>psggrqf</code></p> <p>DOUBLE PRECISION for <code>pdggrqf</code></p> <p>COMPLEX for <code>pcggrqf</code></p> <p>DOUBLE COMPLEX for <code>pzggrqf</code>.</p> <p>Pointer into the local memory to an array of size $(lld_b, LOCc(jb+n-1))$. Contains the local pieces of the p-by-n matrix $\text{sub}(B)$ to be factored.</p>
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.
<i>descb</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .
<i>work</i>	<p>(local)</p> <p>REAL for <code>psggrqf</code></p> <p>DOUBLE PRECISION for <code>pdggrqf</code></p> <p>COMPLEX for <code>pcggrqf</code></p> <p>DOUBLE COMPLEX for <code>pzggrqf</code>.</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) INTEGER.</p> <p>Size of <i>work</i>, must be at least $lwork \geq \max(mb_a * (mpa0 + nqa0 + mb_a), \max((mb_a * (mb_a - 1)) / 2, (ppb0 + nqb0) * mb_a) + mb_a * mb_a, nb_b * (ppb0 + nqb0 + nb_b))$, where</p> <p>$irow = \text{mod}(ia - 1, mb_a)$,</p> <p>$icol = \text{mod}(ja - 1, nb_a)$,</p> <p>$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,</p> <p>$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,</p> <p>$mpa0 = \text{numroc}(m + irow, mb_a, MYROW, iarow, NPROW)$,</p> <p>$nqa0 = \text{numroc}(m + icol, nb_a, MYCOL, iacol, NPCOL)$</p>

```

iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW ),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL ),
ppb0 = numroc (p+iroffb, mb_b, MYROW, ibrow,NPROW),
nqb0 = numroc (n+icoffb, nb_b, MYCOL, ibcol,NPCOL)

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

and `numroc`, `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, if $m \leq n$, the upper triangle of $A(\text{ia}:\text{ia}+m-1, \text{ja}+n-m:\text{ja}+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <code>taua</code> , represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors (see <i>Application Notes</i> below).
<code>taua, taub</code>	<p>(local)</p> <p>REAL for <code>psggqrf</code></p> <p>DOUBLE PRECISION for <code>pdggqrf</code></p> <p>COMPLEX for <code>pcggqrf</code></p> <p>DOUBLE COMPLEX for <code>pzggqrf</code>.</p> <p>Arrays of size $LOCr(\text{ia}+m-1)$ for <code>taua</code> and $LOCc(\text{jb}+\min(p, n)-1)$ for <code>taub</code>.</p> <p>The array <code>taua</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q. <code>taua</code> is tied to the distributed matrix A. (See <i>Application Notes</i> below).</p> <p>The array <code>taub</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z. <code>taub</code> is tied to the distributed matrix B. (See <i>Application Notes</i> below).</p>
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	<p>(global) INTEGER.</p> <p>= 0: the execution is successful.</p>

< 0 : if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(i_a)*H(i_a+1)*...*H(i_a+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau_{au} * v * v'$$

where τ_{au} is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(i_a+m-k+i-1, j_a:j_a+n-k+i-2)$, and τ_{au} in $\tau_{au}(i_a+m-k+i-1)$. To form Q explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungrq](#). To use Q to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmrq](#).

The matrix Z is represented as a product of elementary reflectors

$$Z = H(j_b)*H(j_b+1)*...*H(j_b+k-1), \text{ where } k = \min(p, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_{bu} * v * v'$$

where τ_{bu} is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:p)$ is stored on exit in $B(i_b+i:i_b+p-1, j_b+i-1)$, and τ_{bu} in $\tau_{bu}(j_b+i-1)$. To form Z explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use Z to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Symmetric Eigenproblems

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form T and then find the eigenvalues and eigenvectors of the tridiagonal matrix T . ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table "Computational Routines for Solving Symmetric Eigenproblems"](#).

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the QTQ algorithm, or bisection followed by inverse iteration).

Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	p?sytrd/p?hetrd		
Multiply matrix after reduction		p?ormtr/p?unmtr	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T by a QTQ method			steqr2*
Find selected eigenvalues of a tridiagonal matrix T via bisection			p?stebz

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Find selected eigenvectors of a tridiagonal matrix T by inverse iteration			p?stein

* This routine is described as part of auxiliary ScaLAPACK routines.

[p?sytrd](#)

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
call pssytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Include Files

Description

The `p?sytrd` routine reduces a real symmetric matrix $\text{sub}(A)$ to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q^* \text{sub}(A) Q = T,$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<code>uplo</code>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is stored: If <code>uplo = 'U'</code> , upper triangular If <code>uplo = 'L'</code> , lower triangular
<code>n</code>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pssytrd</code> DOUBLE PRECISION for <code>pdsytrd</code> . Pointer into the local memory to an array of size $(l1d_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the symmetric distributed matrix $\text{sub}(A)$. If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See <i>Application Notes</i> below.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <i>pssytrd</i> DOUBLE PRECISION for <i>pdsytrd</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least: $lwork \geq \max(NB * (np + 1), 3 * NB)$, where $NB = mb_a = nb_a$, $np = \text{numroc}(n, NB, MYROW, iarow, NPROW)$, $iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW)$. <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors. See <i>Application Notes</i> below.
<i>d</i>	(local) REAL for <i>pssytrd</i> DOUBLE PRECISION for <i>pdsytrd</i> . Arrays of size $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix <i>T</i> : $d(i) = A(i,i)$. <i>d</i> is tied to the distributed matrix <i>A</i> .
<i>e</i>	(local) REAL for <i>pssytrd</i> DOUBLE PRECISION for <i>pdsytrd</i> .

Arrays of size $LOCc(ja+n-1)$ if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$e(i) = A(i,i+1)$ if $uplo = 'U'$,

$e(i) = A(i+1,i)$ if $uplo = 'L'$.

e is tied to the distributed matrix A .

tau

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. *tau* is tied to the distributed matrix A .

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a real scalar, and v is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) H(2) \dots H(n-1).$$

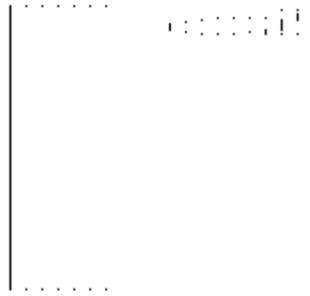
Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

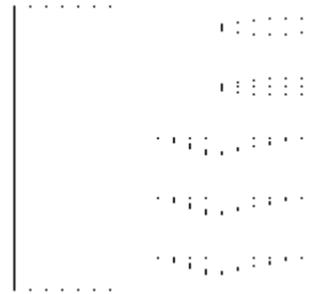
where τ is a real scalar, and v is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of $sub(A)$ on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:



If *uplo* = 'L':



where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *v_i* denotes an element of the vector defining *H(i)*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?sytrd.

Syntax

call psormtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

call pdormtr(*side*, *uplo*, *trans*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *c*, *ic*, *jc*, *descc*, *work*, *lwork*, *info*)

Include Files

Description

This routine overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic:ic+m-1,jc:jc+n-1*) with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	<i>Q</i> *sub(<i>C</i>)	sub(<i>C</i>)* <i>Q</i>
<i>trans</i> = 'T':	<i>Q</i> ^T *sub(<i>C</i>)	sub(<i>C</i>)* <i>Q</i> ^T

where *Q* is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of *nq* elementary reflectors, as returned by p?sytrd.

If $uplo = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $uplo = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>uplo</i>	(global) CHARACTER. = 'U': Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from $p?sytrd$; = 'L': Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from $p?sytrd$
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $sub(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $sub(C)$ ($n \geq 0$).
<i>a</i>	(local) REAL for $psormtr$ DOUBLE PRECISION for $pdormtr$. Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if $side = 'L'$, and $(lld_a, LOCc(ja+n-1))$ if $side = 'R'$. Contains the vectors that define the elementary reflectors, as returned by $p?sytrd$. If $side='L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$; If $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for $psormtr$ DOUBLE PRECISION for $pdormtr$. Array of size of $ltau$ where if $side = 'L'$ and $uplo = 'U'$, $ltau = LOCc(m_a)$,

if *side* = 'L' and *uplo* = 'L', *ltau* = LOCc(*ja+m-2*),

if *side* = 'R' and *uplo* = 'U', *ltau* = LOCc(*n_a*),

if *side* = 'R' and *uplo* = 'L', *ltau* = LOCc(*ja+n-2*).

tau(*i*) must contain the scalar factor of the elementary reflector *H*(*i*), as returned by `p?sytrd`. *tau* is tied to the distributed matrix *A*.

c

(local) REAL for `psormtr`

DOUBLE PRECISION for `pdormtr`.

Pointer into the local memory to an array of size (*lld_c*, LOCc(*jc+n-1*)). Contains the local pieces of the distributed matrix sub (*C*).

ic, jc

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

desc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

REAL for `psormtr`

DOUBLE PRECISION for `pdormtr`.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of *work*, must be at least:

if *uplo* = 'U',

iaa= *ia*; *jaa*= *ja+1*, *icc*= *ic*; *jcc*= *jc*;

else *uplo* = 'L',

iaa= *ia+1*, *jaa*= *ja*;

If *side* = 'L',

icc= *ic+1*; *jcc*= *jc*;

else *icc*= *ic*; *jcc*= *jc+1*;

end if

end if

If *side* = 'L',

mi= *m-1*; *ni*= *n*

lwork ≥ max((*nb_a**(*nb_a-1*))/2, (*nqc0* + *mpc0*)**nb_a*) + *nb_a***nb_a*

else

If *side* = 'R',

mi= *m*; *mi* = *n-1*;

lwork ≥ max((*nb_a**(*nb_a-1*))/2, (*nqc0* + max(*npa0*+numroc(numroc(*ni+icoffc*, *nb_a*, 0, 0, NPCOL), *nb_a*, 0, 0, *lcmq*), *mpc0*))**nb_a*)+ *nb_a***nb_a*

end if

```

where  $lcmq = lcm/NPCOL$  with  $lcm = ilcm(NPROW, NPCOL)$ ,
 $iroffa = \text{mod}(iaa-1, mb\_a)$ ,
 $icoffa = \text{mod}(jaa-1, nb\_a)$ ,
 $iarow = \text{indxg2p}(iaa, mb\_a, MYROW, rsrc\_a, NPROW)$ ,
 $npa0 = \text{numroc}(ni+iroffa, mb\_a, MYROW, iarow, NPROW)$ ,
 $iroffc = \text{mod}(icc-1, mb\_c)$ ,
 $icoffc = \text{mod}(jcc-1, nb\_c)$ ,
 $icrow = \text{indxg2p}(icc, mb\_c, MYROW, rsrc\_c, NPROW)$ ,
 $iccol = \text{indxg2p}(jcc, nb\_c, MYCOL, csrc\_c, NPCOL)$ ,
 $mpc0 = \text{numroc}(mi+iroffc, mb\_c, MYROW, icrow, NPROW)$ ,
 $nqc0 = \text{numroc}(ni+icoffc, nb\_c, MYCOL, iccol, NPCOL)$ ,

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

$ilcm$, indxg2p and numroc are ScaLAPACK tool functions; $MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the subroutine `blacs_gridinfo`. If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p_xerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
call pchetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

```
call pzhetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Include Files

Description

The `p?hetrd` routine reduces a complex Hermitian matrix $\text{sub}(A)$ to Hermitian tridiagonal form T by a unitary similarity transformation:

$$Q^* \text{sub}(A) Q = T$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$.

Input Parameters

<code>uplo</code>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the Hermitian matrix $\text{sub}(A)$ is stored: If <code>uplo = 'U'</code> , upper triangular If <code>uplo = 'L'</code> , lower triangular
<code>n</code>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) COMPLEX for <code>pchetrd</code> DOUBLE COMPLEX for <code>pzhtrd</code> . Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(\text{ja}+n-1))$. On entry, this array contains the local pieces of the Hermitian distributed matrix $\text{sub}(A)$. If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see <i>Application Notes</i> below).
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<code>desca</code>	(global and local) INTEGER array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>work</code>	(local) COMPLEX for <code>pchetrd</code> DOUBLE COMPLEX for <code>pzhtrd</code> . Workspace array of size <code>lwork</code> .
<code>lwork</code>	(local or global) INTEGER, size of <code>work</code> , must be at least: $\text{lwork} \geq \max(\text{NB} * (\text{np} + 1), 3 * \text{NB})$ where $\text{NB} = \text{mb}_a = \text{nb}_a$, $\text{np} = \text{numroc}(n, \text{NB}, \text{MYROW}, \text{iarow}, \text{NPROW})$,

$iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW)$.

indxg2p and numroc are ScaLAPACK tool functions; $MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a	<p>On exit,</p> <p>If $uplo = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array tau, represent the unitary matrix Q as a product of elementary reflectors; if $uplo = 'L'$, the diagonal and first subdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array tau, represent the unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).</p>
d	<p>(local)</p> <p>REAL for <code>pchetrd</code> DOUBLE PRECISION for <code>pzhetrtd</code>.</p> <p>Arrays of size $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix T:</p> $d(i) = A(i,i).$ <p>d is tied to the distributed matrix A.</p>
e	<p>(local)</p> <p>REAL for <code>pchetrd</code> DOUBLE PRECISION for <code>pzhetrtd</code>.</p> <p>Arrays of size $LOCc(ja+n-1)$ if $uplo = 'U'$; $LOCc(ja+n-2)$ - otherwise.</p> <p>The off-diagonal elements of the tridiagonal matrix T:</p> $e(i) = A(i,i+1) \text{ if } uplo = 'U',$ $e(i) = A(i+1,i) \text{ if } uplo = 'L'.$ <p>e is tied to the distributed matrix A.</p>
tau	<p>(local) COMPLEX for <code>pchetrd</code> DOUBLE COMPLEX for <code>pzhetrtd</code>.</p> <p>Array of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A.</p>
$work(1)$	<p>On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.</p>
$info$	<p>(global) INTEGER.</p>

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1)*...*H(2)*H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a complex scalar, and v is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1)*H(2)*...*H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If $uplo = 'L'$:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.

Syntax

```
call pcunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Include Files

Description

This routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $nq-1$ elementary reflectors, as returned by p?hetrd.

If $uplo = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $uplo = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

$side$	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
$trans$	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
$uplo$	(global) CHARACTER. = 'U': Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?hetrd; = 'L': Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?hetrd
m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).

<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>C</i>) ($n \geq 0$).
<i>a</i>	(local) REAL for <i>pcunmtr</i> DOUBLE PRECISION for <i>pzunmtr</i> . Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCc(ja+n-1))$ if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by <i>p?hetrd</i> . If <i>side</i> ='L', $lld_a \geq \max(1, LOCr(ia+m-1))$; If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <i>pcunmtr</i> DOUBLE COMPLEX for <i>pzunmtr</i> . Array of size of <i>ltau</i> where If <i>side</i> = 'L' and <i>uplo</i> = 'U', <i>ltau</i> = $LOCc(m_a)$, if <i>side</i> = 'L' and <i>uplo</i> = 'L', <i>ltau</i> = $LOCc(ja+m-2)$, if <i>side</i> = 'R' and <i>uplo</i> = 'U', <i>ltau</i> = $LOCc(n_a)$, if <i>side</i> = 'R' and <i>uplo</i> = 'L', <i>ltau</i> = $LOCc(ja+n-2)$. <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector <i>H</i> (<i>i</i>), as returned by <i>p?hetrd</i> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for <i>pcunmtr</i> DOUBLE COMPLEX for <i>pzunmtr</i> . Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$. Contains the local pieces of the distributed matrix sub (<i>C</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>desc</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for <i>pcunmtr</i> DOUBLE COMPLEX for <i>pzunmtr</i> .

*lwork*Workspace array of size *lwork*.(local or global) INTEGER, size of *work*, must be at least:

```

If uplo = 'U',
  iaa= ia; jaa= ja+1, icc= ic; jcc= jc;
else uplo = 'L',
  iaa= ia+1, jaa= ja;
If side = 'L',
  icc= ic+1; jcc= jc;
else icc= ic; jcc= jc+1;
end if
end if
If side = 'L',
  mi= m-1; ni= n
  lwork≥max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) + nb_a*nb_a
else
If side = 'R',
  mi= m; mi = n-1;
  lwork≥max((nb_a*(nb_a-1))/2, (nqc0 +
max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a,
0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a
end if
where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
  iroffa = mod(iaa-1, mb_a),
  icoffa = mod(jaa-1, nb_a),
  iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
  npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
  iroffc = mod(icc-1, mb_c),
  icoffc = mod(jcc-1, nb_c),
  icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
  iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
  mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
  nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTEmod(*x*, *y*) is the integer remainder of *x*/*y*.

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`. If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

```
call psstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
            iblock, isplit, work, iwork, liwork, info)
```

```
call pdstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
            iblock, isplit, work, iwork, liwork, info)
```

Include Files

Description

The `p?stebz` routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval `[vl,vu]`, or the eigenvalues indexed `il` through `iu`. A static partitioning of work is done at the beginning of `p?stebz` which results in all processes finding an (almost) equal number of eigenvalues.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>ictxt</i>	(global) INTEGER. The BLACS context handle.
<i>range</i>	(global) CHARACTER. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues in the interval [<i>vl</i> , <i>vu</i>]. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> through <i>iu</i> .
<i>order</i>	(global) CHARACTER. Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	(global) INTEGER. The order of the tridiagonal matrix T ($n \geq 0$).
<i>vl, vu</i>	(global) REAL for <i>psstebz</i> DOUBLE PRECISION for <i>pdstebz</i> . If <i>range</i> = 'V', the routine computes the lower and the upper bounds for the eigenvalues on the interval [<i>l</i> , <i>vu</i>]. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	(global) INTEGER. Constraint: $1 \leq il \leq iu \leq n$. If <i>range</i> = 'I', the index of the smallest eigenvalue is returned for <i>il</i> and of the largest eigenvalue for <i>iu</i> (assuming that the eigenvalues are in ascending order) must be returned. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	(global) REAL for <i>psstebz</i> DOUBLE PRECISION for <i>pdstebz</i> . The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i> . If $abstol \leq 0$, then the tolerance is taken as $ulp T $, where <i>ulp</i> is the machine precision, and $ T $ means the 1-norm of T

Eigenvalues will be computed most accurately when *abstol* is set to the underflow threshold `slamch('U')`, not 0. Note that if eigenvectors are desired later by inverse iteration (`p?stein`), *abstol* should be set to $2 * p?lamch('S')$.

- d* (global)
 REAL for `psstebz`
 DOUBLE PRECISION for `pdstebz`.
 Array of size *n*.
 Contains *n* diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.
- e* (global)
 REAL for `psstebz`
 DOUBLE PRECISION for `pdstebz`.
 Array of size *n* - 1.
 Contains (*n*-1) off-diagonal elements of the tridiagonal matrix *T*. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.
- work* (local)
 REAL for `psstebz`
 DOUBLE PRECISION for `pdstebz`.
 Array of size $\max(5n, 7)$. This is a workspace array.
- lwork* (local) INTEGER. The size of the *work* array must be $\geq \max(5n, 7)$.
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.
- iwork* (local) INTEGER. Array of size $\max(4n, 14)$. This is a workspace array.
- liwork* (local) INTEGER. The size of the *iwork* array must $\geq \max(4n, 14, NPROCS)$.
 If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

- m* (global) INTEGER. The actual number of eigenvalues found. $0 \leq m \leq n$

<i>nsplit</i>	(global) INTEGER. The number of diagonal blocks detected in T . $1 \leq nsplit \leq n$
<i>w</i>	(global) REAL for <i>psstebz</i> DOUBLE PRECISION for <i>pdstebz</i> . Array of size n . On exit, the first m elements of w contain the eigenvalues on all processes.
<i>iblock</i>	(global) INTEGER. Array of size n . At each row/column j where $e(j)$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit <i>iblock</i> (i) specifies which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs to.

NOTE

In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

<i>isplit</i>	(global) INTEGER. Array of size n . Contains the splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), and so on, and the <i>nsplit</i> -th submatrix consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= n . (Only the first <i>nsplit</i> elements are used, but since the <i>nsplit</i> values are not known, n words must be reserved for <i>isplit</i> .)
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if <i>info</i> = - i , the i -th argument has an illegal value. If <i>info</i> > 0, some or all of the eigenvalues fail to converge or are not computed. If <i>info</i> = 1, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances. If <i>info</i> = 2, mismatch between the number of eigenvalues output and the number desired. If <i>info</i> = 3: <i>range</i> ='I', and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating-point arithmetic. Increase the <i>fudge</i> parameter, recompile, and try again.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

```
call psstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pdstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pcstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pzstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)
```

Include Files

Description

The `p?stein` routine computes the eigenvectors of a symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter `lwork`. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `?stein2` (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE

If the eigenvectors obtained are not orthogonal, increase `lwork` and run the code again.

$p = \text{NPROW} * \text{NPCOL}$ is the total number of processes.

Input Parameters

<code>n</code>	(global) INTEGER. The order of the matrix T ($n \geq 0$).
<code>m</code>	(global) INTEGER. The number of eigenvectors to be returned.
<code>d, e, w</code>	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <code>d</code> of size n contains the diagonal elements of T . <code>e</code> of size $n-1$ contains the off-diagonal elements of T . <code>w</code> of size m contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array <code>w</code> from <code>p?stebz</code> with <code>order = 'B'</code> is expected. The array should be replicated in all processes.)
<code>iblock</code>	(global) INTEGER.

Array of size n . The submatrix indices associated with the corresponding eigenvalues in w : 1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array $iblock$ from `p?stebz` is expected here).

isplit

(global) INTEGER.

Array of size n . The splitting points at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, and so on, and the $nsplit$ -th submatrix consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$. (The output array $isplit$ from `p?stebz` is expected here.)

orfac

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within $orfac * ||T||$ of each other are to be orthogonalized. However, if the workspace is insufficient (see *lwork*), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if *orfac* is equal to zero. A default value of 1000 is used if *orfac* is negative. *orfac* should be identical on all processes

iz, jz

(global) INTEGER. The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

descz

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Z .

work

(local). REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Workspace array of size *lwork*.*lwork*

(local) INTEGER.

lwork controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is $nvec = \text{floor}((lwork - \max(5*n, np00*mq00))/n)$. Eigenvectors corresponding to eigenvalue clusters of size $(nvec - \text{ceil}(m/p) + 1)$ are guaranteed to be orthogonal (the orthogonality is similar to that obtained from `?stein2`).

NOTE

lwork must be no smaller than $\max(5*n, np00*mq00) + \text{ceil}(m/p) * n$ and should have the same input value on all processes.

It is the minimum value of *lwork* input on different processes that is significant.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

iwork (local) INTEGER.

Workspace array of size $3n+p+1$.

liwork (local) INTEGER. The size of the array *iwork*. It must be greater than $3*n+p+1$.

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

z (local)

REAL for `psstein`

DOUBLE PRECISION for `pdstein`

COMPLEX for `pcstein`

DOUBLE COMPLEX for `pzstein`.

Array of size $descz(dlen_), n/NPCOL + NB$. *z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See [?stein2](#)). On output, *z* is distributed across the *p* processes in block cyclic format.

work(1)[0] On exit, *work(1)* gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork On exit, *iwork(1)* contains the amount of integer workspace required.

On exit, the *iwork(2)* through *iwork(p+2)* indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork(i+2)+1* through *iwork(i+3)*.

ifail (global) INTEGER. Array of size *m*. On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in [?stein](#)), then $info > 0$ is returned. If $\text{mod}(info, m+1) > 0$, then for $i=1$ to $\text{mod}(info, m+1)$, the eigenvector corresponding to the eigenvalue $w(ifail(i))$ failed to converge (*w* refers to the array of eigenvalues on output).

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

iclustr (global) INTEGER. Array of size $2*p$.

This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*I-1) to *iclustr*(2*I), *i* = 1 to *info*/(*m*+1), could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr* is a zero terminated array: *iclustr*(2*k) ≠ 0 and *iclustr*(2*k+1) = 0 if and only if *k* is the number of clusters.

gap

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The *info*/*m* output values in this array correspond to the *info*/(*m*+1) clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(O(n) * macheps) / gap(i)$.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*),

If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0: if mod(*info*, *m*+1) = *i*, then *i* eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. If *info*/(*m*+1) = *i*, then eigenvectors corresponding to *i* clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Nonsymmetric Eigenvalue Problems

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table "Computational Routines for Solving Nonsymmetric Eigenproblems" lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr / p?unmhr	

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Find eigenvalues and Schur factorization			p?lahqr , p?hseqr

p?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

```
call psgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

Include Files

Description

The `p?gehrd` routine reduces a real/complex general distributed matrix $\text{sub}(A)$ to upper Hessenberg form H by an orthogonal or unitary similarity transformation

$$Q^* \text{sub}(A) Q = H,$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$.

Input Parameters

<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. It is assumed that $\text{sub}(A)$ is already upper triangular in rows $\text{ia}:\text{ia}+\text{ilo}-2$ and $\text{ia}+\text{ihi}:\text{ia}+n-1$ and columns $\text{ja}:\text{ja}+\text{ilo}-2$ and $\text{ja}+\text{ihi}:\text{ja}+n-1$. (See <i>Application Notes</i> below). If $n > 0$, $1 \leq \text{ilo} \leq \text{ihi} \leq n$; otherwise set $\text{ilo} = 1$, $\text{ihi} = n$.
<i>a</i>	(local) REAL for <code>psgehrd</code> DOUBLE PRECISION for <code>pdgehrd</code> COMPLEX for <code>pcgehrd</code> DOUBLE COMPLEX for <code>pzgehrd</code> . Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(\text{ja}+n-1))$. On entry, this array contains the local pieces of the n -by- n general distributed matrix $\text{sub}(A)$ to be reduced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size dlen_- . The array descriptor for the distributed matrix A .
<i>work</i>	(local)

REAL for psgehrd
 DOUBLE PRECISION for pdgehrd
 COMPLEX for pcgehrd
 DOUBLE COMPLEX for pzgehrd.
 Workspace array of size *lwork*.

lwork

(local or global) INTEGER, size of the array *work*. *lwork* is local input and must be at least

$$lwork \geq NB * NB + NB * \max(ihip+1, ihlp+inlq)$$

where $NB = mb_a = nb_a$,

$$iroffa = \text{mod}(ia-1, NB),$$

$$icoffa = \text{mod}(ja-1, NB),$$

$$ioff = \text{mod}(ia+ilo-2, NB), \quad iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW), \quad ihip = \text{numroc}(ihi+iroffa, NB, MYROW, iarow, NPROW),$$

$$ilrow = \text{indxg2p}(ia+ilo-1, NB, MYROW, rsrc_a, NPROW),$$

$$ihlp = \text{numroc}(ihi-ilo+ioff+1, NB, MYROW, ilrow, NPROW),$$

$$ilcol = \text{indxg2p}(ja+ilo-1, NB, MYCOL, csrc_a, NPCOL),$$

$$inlq = \text{numroc}(n-ilo+ioff+1, NB, MYCOL, ilcol, NPCOL),$$
NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters*a*

On exit, the upper triangle and the first subdiagonal of `sub(A)` are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau

(local). REAL for psgehrd

DOUBLE PRECISION for pdgehrd

COMPLEX for pcgehrd

DOUBLE COMPLEX for pzgehrd.

Array of size at least $\max(ja+n-2)$.

The scalar factors of the elementary reflectors (see *Application Notes* below). Elements $ja:ja+ilo-2$ and $ja+ihi:ja+n-2$ of the global vector τ are set to zero. τ is tied to the distributed matrix A .

$work(1)$

On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$info$

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo)*H(ilo+1)*...*H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(ihi+1:n) = 0$; $v(i+2:ihi)$ is stored on exit in $a(ia+ilo+i:ia+ihi-1, ja+ilo+i-2)$, and τ in $\tau(ja+ilo+i-2)$. The contents of

$a(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & a & a & a & a & a \\ & & & a & a & a & a \\ & & & & a & a & a \\ & & & & & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & a & h & h & h & a \\ & a & h & h & h & h & a \\ & & h & h & h & h & h \\ v_2 & & h & h & h & h & h \\ v_2 & v_3 & & h & h & h & h \\ v_2 & v_3 & v_4 & & h & h & h \\ & & & & & & a \end{bmatrix}$$

where a denotes an element of the original matrix $\text{sub}(A)$, H denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+i\text{ilo}+i-2)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
call psormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Include Files

Description

The p?ormhr routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by p?gehrd.

$Q = H(ilo) H(ilo+1) \dots H(ihi-1)$.

Input Parameters

$side$ (global) CHARACTER
 $= 'L'$: Q or Q^T is applied from the left.

	= 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (C) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (C) ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER. <i>ilo</i> and <i>ihi</i> must have the same values as in the previous call of <code>p?gehrd</code> . Q is equal to the unit matrix except for the distributed submatrix $Q(ia + ilo:ia + ihi - 1, ja + ilo:ja + ihi - 1)$. If <i>side</i> = 'L', $1 \leq ilo \leq ihi \leq \max(1, m)$; If <i>side</i> = 'R', $1 \leq ilo \leq ihi \leq \max(1, n)$; <i>ilo</i> and <i>ihi</i> are relative indexes.
<i>a</i>	(local) REAL for <code>psormhr</code> DOUBLE PRECISION for <code>pdormhr</code> Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCc(ja+n-1))$ if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by <code>p?gehrd</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for <code>psormhr</code> DOUBLE PRECISION for <code>pdormhr</code> Array of size $LOCc(ja+m-2)$ if <i>side</i> = 'L', and $LOCc(ja+n-2)$ if <i>side</i> = 'R'. <i>tau(j)</i> contains the scalar factor of the elementary reflector $H(j)$ as returned by <code>p?gehrd</code> . <i>tau</i> is tied to the distributed matrix A .
<i>c</i>	(local) REAL for <code>psormhr</code> DOUBLE PRECISION for <code>pdormhr</code>

Pointer into the local memory to an array of size $(lld_c, LOCC(jc+n-1))$.
Contains the local pieces of the distributed matrix sub(C).

ic, jc (global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.

desc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix C.

work (local)
REAL for psormhr
DOUBLE PRECISION for pdormhr
Workspace array with size *lwork*.

lwork (local or global) INTEGER.
The size of the array *work*.
lwork must be at least $iaa = ia + ilo; jaa = ja + ilo - 1;$
If *side* = 'L',
 $mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc;$
 $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$
else if *side* = 'R',
 $mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo;$
 $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(ni + icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$
end if
where $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL),$
 $irow = \text{indxg2p}(iaa, mb_a, MYROW, rsrc_a, NPROW),$
 $npa0 = \text{numroc}(ni + irow, mb_a, MYROW, rsrc_a, NPROW),$
 $irow = \text{indxg2p}(icc, mb_c, MYROW, rsrc_c, NPROW),$
 $icrow = \text{indxg2p}(jcc, nb_c, MYCOL, csrc_c, NPCOL),$
 $mpc0 = \text{numroc}(mi + icrow, mb_c, MYROW, icrow, NPROW),$
 $nqc0 = \text{numroc}(ni + icrow, nb_c, MYCOL, icrow, NPCOL),$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	<code>sub(C)</code> is overwritten by $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
call pcunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pzunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Include Files

Description

This routine overwrites the general complex distributed *m*-by-*n* matrix $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$ with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q*\text{sub}(C)$	$\text{sub}(C)*Q$
<code>trans = 'H':</code>	$Q^H*\text{sub}(C)$	$\text{sub}(C)*Q^H$

where *Q* is a complex unitary distributed matrix of order *nq*, with $nq = m$ if `side = 'L'` and $nq = n$ if `side = 'R'`.

Q is defined as the product of *ihi-ilo* elementary reflectors, as returned by `p?gehrd`.

$Q = H(\text{ilo}) H(\text{ilo}+1) \dots H(\text{ihi}-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (C) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (C) ($n \geq 0$).
<i>ilo, ihi</i>	(global) INTEGER These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <code>p?gehrd</code> . Q is equal to the unit matrix except in the distributed submatrix $Q(ia+ilo:ia+ihi-1, ja+ilo:ja+ihi-1)$. If <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq \max(1, m)$. If <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq \max(1, n)$ <i>ilo</i> and <i>ihi</i> are relative indexes.
<i>a</i>	(local) COMPLEX for <code>pcunmhr</code> DOUBLE COMPLEX for <code>pzunmhr</code> . Pointer into the local memory to an array of size $(lld_a, LOCc(ja+m-1))$ if <i>side</i> = 'L', and $(lld_a, LOCc(ja+n-1))$ if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by <code>p?gehrd</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) COMPLEX for <code>pcunmhr</code> DOUBLE COMPLEX for <code>pzunmhr</code> . Array of size $LOCc(ja+m-2)$, if <i>side</i> = 'L', and $LOCc(ja+n-2)$ if <i>side</i> = 'R'. <i>tau(j)</i> contains the scalar factor of the elementary reflector $H(j)$ as returned by <code>p?gehrd</code> . <i>tau</i> is tied to the distributed matrix A .

<i>c</i>	<p>(local)</p> <p>COMPLEX for pcunmhr</p> <p>DOUBLE COMPLEX for pzunmhr.</p> <p>Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$.</p> <p>Contains the local pieces of the distributed matrix sub(C).</p>
<i>ic, jc</i>	<p>(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.</p>
<i>desc</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix C.</p>
<i>work</i>	<p>(local)</p> <p>COMPLEX for pcunmhr</p> <p>DOUBLE COMPLEX for pzunmhr.</p> <p>Workspace array with size <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global)</p> <p>The size of the array <i>work</i>.</p> <p><i>lwork</i> must be at least $iaa = ia + ilo; jaa = ja + ilo - 1;$</p> <p>If <i>side</i> = 'L', $mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc;$ $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$</p> <p>else if <i>side</i> = 'R',</p> <p>$mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo;$ $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 +$ $\max(npa0 + \text{numroc}(\text{numroc}(ni + icoffc, nb_a, 0, 0, NPCOL), nb_a,$ $0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$</p> <p>end if</p> <p>where $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL),$ $irow = \text{mod}(iaa - 1, mb_a),$ $icoffa = \text{mod}(jaa - 1, nb_a),$ $iarow = \text{indxg2p}(iaa, mb_a, MYROW, rsrc_a, NPROW),$ $npa0 = \text{numroc}(ni + irow, mb_a, MYROW, iarow, NPROW),$ $irow = \text{mod}(icc - 1, mb_c),$ $icoffc = \text{mod}(jcc - 1, nb_c),$ $icrow = \text{indxg2p}(icc, mb_c, MYROW, rsrc_c, NPROW),$ $iccol = \text{indxg2p}(jcc, nb_c, MYCOL, csrc_c, NPCOL),$ $mpc0 = \text{numroc}(mi + icrow, mb_c, MYROW, icrow, NPROW),$ $nqc0 = \text{numroc}(ni + icoffc, nb_c, MYCOL, iccol, NPCOL),$</p>

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?erbla`.

Output Parameters

<code>c</code>	<code>C</code> is overwritten by $Q^* \text{sub}(C)$ or $Q'^* \text{sub}(C)$ or $\text{sub}(C) * Q'$ or $\text{sub}(C) * Q$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

```
call pslahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info)
```

```
call pdlahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info)
```

Include Files**Description**

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns `ilo` and `ihi`.

Input Parameters

<code>wantt</code>	(global) LOGICAL If <code>wantt = .TRUE.</code> , the full Schur form T is required; If <code>wantt = .FALSE.</code> , only eigenvalues are required.
--------------------	---

<i>wantz</i>	(global) LOGICAL. If <i>wantz</i> = .TRUE., the matrix of Schur vectors <i>Z</i> is required; If <i>wantz</i> = .FALSE., Schur vectors are not required.
<i>n</i>	(global) INTEGER. The order of the Hessenberg matrix <i>A</i> (and <i>z</i> if <i>wantz</i>). $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER. It is assumed that <i>A</i> is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $A(ilo, ilo-1) = 0$ (unless $ilo = 1$). <i>p?lahqr</i> works primarily with the Hessenberg submatrix in rows and columns <i>ilo</i> to <i>ihi</i> , but applies transformations to all of <i>H</i> if <i>wantt</i> is .TRUE.. $1 \leq ilo \leq \max(1, ihi); ihi \leq n$.
<i>a</i>	(global) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Array, of size $(lld_a, *)$. On entry, the upper Hessenberg matrix <i>A</i> .
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iloz, ihiz</i>	(global) INTEGER. Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ilo;$ $ihi \leq ihiz \leq n$.
<i>z</i>	(global)REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Array. If <i>wantz</i> is .TRUE., on entry <i>z</i> must contain the current matrix <i>Z</i> of transformations accumulated by <i>pdhseqr</i> . If <i>wantz</i> is .FALSE., <i>z</i> is not referenced.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local) REAL for <i>pslahqr</i> DOUBLE PRECISION for <i>pdlahqr</i> Workspace array with size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The size of <i>work</i> . <i>lwork</i> is assumed big enough so that $lwork \geq 3*n + \max(2*\max(lld_z, lld_a) + 2*LOCq(n), 7*ceil(n/hbl)/lcm(NPROW, NPCOL))$. If <i>lwork</i> = -1, then <i>work</i> (1) gets set to the above number and the code returns immediately.
<i>iwork</i>	(global and local) INTEGER array of size <i>ilwork</i> .
<i>ilwork</i>	(local) INTEGER This holds some of the <i>iblk</i> integer arrays.

Output Parameters

<i>a</i>	On exit, if <i>wantt</i> is <code>.TRUE.</code> , <i>A</i> is upper quasi-triangular in rows and columns <i>ilo:ihi</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is <code>.FALSE.</code> , the contents of <i>A</i> are unspecified on exit.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>wr, wi</i>	(global replicated output) REAL for <code>pshahqr</code> DOUBLE PRECISION for <code>pdlahqr</code> Arrays of size <i>n</i> each. The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)-th, with <i>wi</i> (<i>i</i>) > 0 and <i>wi</i> (<i>i</i> +1) < 0. If <i>wantt</i> is <code>.TRUE.</code> , the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>A</i> . <i>A</i> may be returned with larger diagonal blocks until the next release.
<i>z</i>	On exit <i>z</i> has been updated; transformations are applied only to the submatrix <i>Z</i> (<i>iloz:ihiz, ilo:ihi</i>).
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: the parameter number - <i>info</i> is incorrect or inconsistent > 0: <code>p?lahqr</code> failed to compute all the eigenvalues <i>ilo</i> to <i>ihi</i> in a total of $30 * (ihi - ilo + 1)$ iterations; if <i>info</i> = <i>i</i> , elements <i>i</i> +1: <i>ihi</i> of <i>wr</i> and <i>wi</i> contain the eigenvalues that have been successfully computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?hseqr

Computes eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
call pshseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
```

```
call pdhseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
```

Include Files

Description

`p?hseqr` computes the eigenvalues of an upper Hessenberg matrix *H* and, optionally, the matrices *T* and *Z* from the Schur decomposition $H = Z * T * Z^T$, where *T* is an upper quasi-triangular matrix (the Schur form), and *Z* is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q : $A = Q*H*Q^T = (QZ)*T*(QZ)^T$.

Input Parameters

<i>job</i>	(global) CHARACTER*1 = 'E': compute eigenvalues only; = 'S': compute eigenvalues and the Schur form T.
<i>compz</i>	(global) CHARACTER*1 = 'N': no Schur vectors are computed; = 'I': z is initialized to the unit matrix and the matrix Z of Schur vectors of H is returned; = 'V': z must contain an orthogonal matrix Q on entry, and the product $Q*Z$ is returned.
<i>n</i>	(global) INTEGER The order of the Hessenberg matrix H . $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$. ilo and ihi are normally set by a previous call to p?gebal , and then passed to p?gehrd when the matrix output by p?gebal is reduced to Hessenberg form. Otherwise ilo and ihi should be set to 1 and n respectively. If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.
<i>h</i>	REAL for pshseqr DOUBLE PRECISION for pdhseqr (global) array of size (<i>desc</i> (<i>lld_</i>), <i>LOC</i> _c (<i>n</i>)) The upper Hessenberg matrix H .
<i>desc</i>	(global and local) INTEGER array of size <i>dlen_</i> The array descriptor for the distributed matrix H .
<i>z</i>	REAL for pshseqr DOUBLE PRECISION for pdhseqr (global) array If <i>compz</i> = 'V', on entry z must contain the current matrix Z of accumulated transformations from, for example, p?gehrd . If <i>compz</i> = 'I', on entry z need not be set.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> The array descriptor for the distributed matrix z .
<i>work</i>	REAL for pshseqr

DOUBLE PRECISION for `pdhseqr`
 (local workspace) array of size `lwork`.

`lwork` (local) INTEGER
 The length of the workspace array `work`.

`iwork` (local workspace) INTEGER array of size `liwork`

`liwork` (local) INTEGER
 The length of the workspace array `iwork`.

OUTPUT Parameters

`h` If `job = 'S'`, H is upper quasi-triangular in rows and columns `ilo:ihi`, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $h(i,i) = h(i+1,i+1)$ and $h(i+1,i)*h(i,i+1) < 0$. If `info = 0` and `job = 'E'`, the contents of `h` are unspecified on exit.

`wr, wi` REAL for `pshseqr`
 DOUBLE PRECISION for `pdhseqr`
 (global) array of size `n`
 The real and imaginary parts, respectively, of the computed eigenvalues `ilo` to `ihi` are stored in the corresponding elements of `wr` and `wi`. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of `wr` and `wi`, say the i -th and $(i+1)$ -th, with $wi(i) > 0$ and $wi(i+1) < 0$. If `job = 'S'`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in `h`.

`z` REAL for `pshseqr`
 DOUBLE PRECISION for `pdhseqr`
 (global) array
`z` is updated; transformations are applied only to the submatrix `z(ilo:ihi,ilo:ihi)`.
 If `compz = 'N'`, `z` is not referenced.
 If `compz = 'I'` and `info = 0`, `z` contains the orthogonal matrix Z of the Schur vectors of H .

`info` INTEGER
 = 0: successful exit
 < 0: if `info = -i`, the i -th argument had an illegal value (see also below for -7777 and -8888).
 > 0: if `info = i`, `pshseqr` failed to compute all of the eigenvalues. Elements 1:`ilo-1` and `i+1:n` of `wr` and `wi` contain those eigenvalues which have been successfully computed. (Failures are rare.)

If $info > 0$ and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final, output value of H .

If $info > 0$ and $job = 'S'$, then on exit $(*)$ (initial value of H)* $U = U$ *(final value of H) where U is an orthogonal matrix. The final value of H is upper Hessenberg and quasi-triangular in rows and columns $info+1$ through ihi .

If $info > 0$ and $compz = 'V'$, then on exit (final value of Z) = (initial value of Z)* U where U is the orthogonal matrix in $(*)$ (regardless of the value of job .)

If $info > 0$ and $compz = 'I'$, then on exit (final value of Z) = U where U is the orthogonal matrix in $(*)$ (regardless of the value of job .)

If $info > 0$ and $compz = 'N'$, then z is not accessed.

= -7777: `p laqr0` failed to converge and `p laqr1` was called instead.

= -8888: `p?laqr1` failed to converge and `p?laqr0` was called instead.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Singular Value Decomposition

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general m -by- n matrix A (see "[Singular Value Decomposition](#)" in LAPACK chapter).

To find the SVD of a general matrix A , this matrix is first reduced to a bidiagonal matrix B by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of B is computed using the LAPACK routine `?bdsqr`.

Table "[Computational Routines for Singular Value Decomposition \(SVD\)](#)" lists ScaLAPACK computational routines for performing this decomposition.

Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce A to a bidiagonal matrix	<code>p?gebrd</code>	
Multiply matrix after reduction		<code>p?ormbr/p?unmbr</code>

`p?gebrd`

Reduces a general matrix to bidiagonal form.

Syntax

```
call psgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pdgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pcgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pzgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Include Files

Description

The `p?gebrd` routine reduces a real/complex general m -by- n distributed matrix $sub(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q^*sub(A)P = B.$$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(A) ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(A) ($n \geq 0$).
<i>a</i>	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Real pointer into the local memory to an array of size $(lld_a, LOCC(ja + n - 1))$. On entry, this array contains the distributed matrix sub (A).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>work</i>	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, size of <i>work</i> , must be at least: $lwork \geq nb * (mpa0 + nqa0 + 1) + nqa0$ where $nb = mb_a = nb_a$, $irow = mod(ia - 1, nb)$, $icoffa = mod(ja - 1, nb)$, $iarow = indxg2p(ia, nb, MYROW, rsrc_a, NPROW)$, $iacol = indxg2p(ja, nb, MYCOL, csrc_a, NPCOL)$, $mpa0 = numroc(m + irow, nb, MYROW, iarow, NPROW)$, $nqa0 = numroc(n + icoffa, nb, MYCOL, iacol, NPCOL)$,

NOTE

$mod(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	<p>On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix B; the elements below the diagonal, with the array <code>tauq</code>, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array <code>taup</code>, represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B; the elements below the first subdiagonal, with the array <code>tauq</code>, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array <code>taup</code>, represent the orthogonal matrix P as a product of elementary reflectors. See <i>Application Notes</i> below.</p>
<code>d</code>	<p>(local)</p> <p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$ and $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B: $d(i) = a(i, i)$.</p> <p><code>d</code> is tied to the distributed matrix A.</p>
<code>e</code>	<p>(local)</p> <p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Array of size $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B:</p> <p>If $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$; if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$. <code>e</code> is tied to the distributed matrix A.</p>
<code>tauq, taup</code>	<p>(local)</p> <p>REAL for <code>psgebrd</code></p> <p>DOUBLE PRECISION for <code>pdgebrd</code></p> <p>COMPLEX for <code>pcgebrd</code></p> <p>DOUBLE COMPLEX for <code>pzgebrd</code>.</p>

Arrays of size $LOCc(ja+\min(m,n)-1)$ for $tauq$ and $LOCr(ia+\min(m,n)-1)$ for $taup$. Contain the scalar factors of the elementary reflectors that represent the orthogonal/unitary matrices Q and P , respectively. $tauq$ and $taup$ are tied to the distributed matrix A . See *Application Notes* below.

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1)*H(2)*...*H(n), \text{ and } P = G(1)*G(2)*...*G(n-1).$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - tauq * v * v' \text{ and } G(i) = I - taup * u * u'$$

where $tauq$ and $taup$ are real/complex scalars, and v and u are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

$tauq$ is stored in $tauq(ja+i-1)$ and $taup$ in $taup(ia+i-1)$.

If $m < n$,

$$Q = H(1)*H(2)*...*H(m-1), \text{ and } P = G(1)*G(2)*...*G(m)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - tauq * v * v' \text{ and } G(i) = I - taup * u * u'$$

here $tauq$ and $taup$ are real/complex scalars, and v and u are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

$tauq$ is stored in $tauq(ja+i-1)$ and $taup$ in $taup(ia+i-1)$.

The contents of sub(A) on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

call psormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)

call pdormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)

Include Files

Description

If *vect* = 'Q', the p?ormbr routine overwrites the general real distributed m -by- n matrix $sub(C) = C(ic:ic+m-1, jc: jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q \ sub(C)$	$sub(C) \ Q$
<i>trans</i> = 'T':	$Q^T \ sub(C)$	$sub(C) \ Q^T$

If *vect* = 'P', the routine overwrites $sub(C)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$P \text{ sub}(C)$	$\text{sub}(C) P$
<i>trans</i> = 'T':	$P^T \text{ sub}(C)$	$\text{sub}(C) P^T$

Here Q and P^T are the orthogonal distributed matrices determined by `p?gebrd` when reducing a real distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q*B*P^T$. Q and P^T are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if *side* = 'L' and $nq = n$ if *side* = 'R'. Therefore nq is the order of the orthogonal matrix Q or P^T that is applied.

If *vect* = 'Q', $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If *vect* = 'P', $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) CHARACTER. If <i>vect</i> = 'Q', then Q or Q^T is applied. If <i>vect</i> = 'P', then P or P^T is applied.
<i>side</i>	(global) CHARACTER. If <i>side</i> = 'L', then Q or Q^T , P or P^T is applied from the left. If <i>side</i> = 'R', then Q or Q^T , P or P^T is applied from the right.
<i>trans</i>	(global) CHARACTER. If <i>trans</i> = 'N', no transpose, Q or P is applied. If <i>trans</i> = 'T', then Q^T or P^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (C).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (C).
<i>k</i>	(global) INTEGER. If <i>vect</i> = 'Q', the number of columns in the original distributed matrix reduced by <code>p?gebrd</code> ; If <i>vect</i> = 'P', the number of rows in the original distributed matrix reduced by <code>p?gebrd</code> . Constraints: $k \geq 0$.
<i>a</i>	(local) REAL for <code>psormbr</code> DOUBLE PRECISION for <code>pdormbr</code> .

Pointer into the local memory to an array of size $(lld_a, LOcc(ja + \min(nq, k) - 1))$ if $vect = 'Q'$, and $(lld_a, LOcc(ja + nq - 1))$ if $vect = 'P'$.

$nq = m$ if $side = 'L'$, and $nq = n$ otherwise.

The vectors that define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P , as returned by `p?gebrd`.

If $vect = 'Q'$, $lld_a \geq \max(1, LOCr(ia + nq - 1))$;

If $vect = 'P'$, $lld_a \geq \max(1, LOCr(ia + \min(nq, k) - 1))$.

ia, ja

(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

tau

(local)

REAL for `psormbr`

DOUBLE PRECISION for `pdormbr`.

Array of size $LOcc(ja + \min(nq, k) - 1)$, if $vect = 'Q'$, and $LOCr(ia + \min(nq, k) - 1)$, if $vect = 'P'$.

$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$ or $G(i)$

which determines Q or P , as returned by `pdgebrd` in its array argument $tauq$ or $taup$. tau is tied to the distributed matrix A .

c

(local) REAL for `psormbr`

DOUBLE PRECISION for `pdormbr`

Pointer into the local memory to an array of size $(lld_c, LOcc(jc + n - 1))$.

Contains the local pieces of the distributed matrix sub (C).

ic, jc

(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.

desc

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local)

REAL for `psormbr`

DOUBLE PRECISION for `pdormbr`.

Workspace array of size $lwork$.

lwork

(local or global) INTEGER, size of $work$, must be at least:

If $side = 'L'$

$nq = m$;

```

if ((vect = 'Q' and nq>=k) or (vect is not equal to 'Q' and nq>k)),
  iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;
else
  iaa= ia+1; jaa=ja; mi=m-1; ni=n; icc=ic+1; jcc= jc;
end if
else
  If side = 'R', nq = n;
  if((vect = 'Q' and nq>=k) or (vect is not equal to 'Q' and
  nq>k)),
  iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;
else
  iaa= ia; jaa= ja+1; mi= m; ni= n-1; icc= ic; jcc= jc+1;
end if
end if
If vect = 'Q',
If side = 'L', lwork>=max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
nb_a * nb_a
else if side = 'R',
  lwork>=max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0 +
  numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0,
  lcmq), mpc0))*nb_a) + nb_a*nb_a
end if
else if vect is not equal to 'Q', if side = 'L',
  lwork>=max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
  numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
  lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
  lwork>=max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a
end if
end if
where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with lcm =
ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),

```

```

icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?xerbla`.

Output Parameters

<code>c</code>	On exit, if <code>vect='Q'</code> , <code>sub(C)</code> is overwritten by $Q*\text{sub}(C)$, or $Q'^*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$; if <code>vect='P'</code> , <code>sub(C)</code> is overwritten by $P*\text{sub}(C)$, or $P'^*\text{sub}(C)$, or $\text{sub}(C)*P$, or $\text{sub}(C)*P'$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
call pcunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pzunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

Include Files

Description

If $vect = 'Q'$, the `p?unmbr` routine overwrites the general complex distributed m -by- n matrix $sub(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q*sub(C)$	$sub(C)*Q$
$trans = 'C':$	$Q^H*sub(C)$	$sub(C)*Q^H$

If $vect = 'P'$, the routine overwrites $sub(C)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$P*sub(C)$	$sub(C)*P$
$trans = 'C':$	$P^H*sub(C)$	$sub(C)*P^H$

Here Q and P^H are the unitary distributed matrices determined by `p?gebrd` when reducing a complex distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q*B*P^H$.

Q and P^H are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$. Therefore nq is the order of the unitary matrix Q or P^H that is applied.

If $vect = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $vect = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

$vect$	(global) CHARACTER. If $vect = 'Q'$, then Q or Q^H is applied. If $vect = 'P'$, then P or P^H is applied.
$side$	(global) CHARACTER. If $side = 'L'$, then Q or Q^H , P or P^H is applied from the left. If $side = 'R'$, then Q or Q^H , P or P^H is applied from the right.
$trans$	(global) CHARACTER. If $trans = 'N'$, no transpose, Q or P is applied. If $trans = 'C'$, conjugate transpose, Q^H or P^H is applied.
m	(global) INTEGER. The number of rows in the distributed matrix $sub(C)$ $m \geq 0$.
n	(global) INTEGER. The number of columns in the distributed matrix $sub(C)$ $n \geq 0$.
k	(global) INTEGER.

If *vect* = 'Q', the number of columns in the original distributed matrix reduced by *p?gebrd*;

If *vect* = 'P', the number of rows in the original distributed matrix reduced by *p?gebrd*.

Constraints: $k \geq 0$.

a

(local)

COMPLEX for *psormbr*

DOUBLE COMPLEX for *pdormbr*.

Pointer into the local memory to an array of size $(l1d_a, LOCc(ja + \min(nq, k) - 1))$ if *vect*='Q', and $(l1d_a, LOCc(ja+nq-1))$ if *vect* = 'P'.

$nq = m$ if *side* = 'L', and $nq = n$ otherwise.

The vectors that define the elementary reflectors *H(i)* and *G(i)*, whose products determine the matrices *Q* and *P*, as returned by *p?gebrd*.

If *vect* = 'Q', $l1d_a \geq \max(1, LOCr(ia+nq-1))$;

If *vect* = 'P', $l1d_a \geq \max(1, LOCr(ia+\min(nq, k)-1))$.

ia, ja

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau

(local)

COMPLEX for *pcunmbr*

DOUBLE COMPLEX for *pzunmbr*.

Array of size $LOCc(ja+\min(nq, k)-1)$, if *vect* = 'Q', and $LOCr(ia + \min(nq, k) - 1)$, if *vect* = 'P'.

tau(i) must contain the scalar factor of the elementary reflector *H(i)* or *G(i)*, which determines *Q* or *P*, as returned by *p?gebrd* in its array argument *tauq* or *taup*. *tau* is tied to the distributed matrix *A*.

c

(local) COMPLEX for *pcunmbr*

DOUBLE COMPLEX for *pzunmbr*

Pointer into the local memory to an array of size $(l1d_c, LOCc(jc+n-1))$.

Contains the local pieces of the distributed matrix sub (*C*).

ic, jc

(global) INTEGER. The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

lwork

COMPLEX for pcunmbr

DOUBLE COMPLEX for pzunmbr.

Workspace array of size *lwork*.(local or global) INTEGER, size of *work*, must be at least:If *side* = 'L'*nq* = *m*;if ((*vect* = 'Q' and $nq \geq k$) or (*vect* is not equal to 'Q' and $nq > k$)), *iaa*= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;

else

iaa= *ia*+1; *jaa*= *ja*; *mi*= *m*-1; *ni*= *n*; *icc*= *ic*+1; *jcc*= *jc*;

end if

else

If *side* = 'R', *nq* = *n*;if ((*vect* = 'Q' and $nq \geq k$) or (*vect* is not equal to 'Q' and $nq > k$)),*iaa*= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;

else

iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;

end if

end if

If *vect* = 'Q',If *side* = 'L', $lwork \geq \max((nb_a*(nb_a-1))/2, (nqc0+mpc0)*nb_a) + nb_a*nb_a$ else if *side* = 'R', $lwork \geq \max((nb_a*(nb_a-1))/2, (nqc0 + \max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a$

end if

else if *vect* is not equal to 'Q',if *side* = 'L', $lwork \geq \max((mb_a*(mb_a-1))/2, (mpc0 + \max(mqa0+numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmq), nqc0))*mb_a) + mb_a*mb_a$ else if *side* = 'R', $lwork \geq \max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a$

end if

end if

where *lcmp* = *lcm*/NPROW, *lcmq* = *lcm*/NPCOL, with *lcm* = *ilcm*(NPROW, NPCOL),

```

iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x, y)` is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	On exit, if <code>vect='Q'</code> , <code>sub(C)</code> is overwritten by $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$; if <code>vect='P'</code> , <code>sub(C)</code> is overwritten by $P*\text{sub}(C)$, or $P'*\text{sub}(C)$, or $\text{sub}(C)*P$, or $\text{sub}(C)*P'$.
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Generalized Symmetric-Definite Eigen Problems

This section describes ScaLAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see [Generalized Symmetric-Definite Eigenvalue Problems](#) in LAPACK chapters) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScaLAPACK routines described earlier in this chapter (see [Symmetric Eigenproblems](#)).

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists these routines.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pssygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

```
call pdsygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

Include Files

Description

The p?sygst routine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $ibtype = 1$, the problem is

$$\text{sub}(A)*x = \lambda*\text{sub}(B)*x,$$

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^T)*\text{sub}(A)*\text{inv}(U)$, or $\text{inv}(L)*\text{sub}(A)*\text{inv}(L^T)$.

If $ibtype = 2$ or 3 , the problem is

$$\text{sub}(A)*\text{sub}(B)*x = \lambda*x, \text{ or } \text{sub}(B)*\text{sub}(A)*x = \lambda*x,$$

and $\text{sub}(A)$ is overwritten by $U*\text{sub}(A)*U^T$, or $L^T*\text{sub}(A)*L$.

$\text{sub}(B)$ must have been previously factorized as U^T*U or $L*L^T$ by p?potrf.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If $ibtype = 1$, compute $\text{inv}(U^T)*\text{sub}(A)*\text{inv}(U)$, or $\text{inv}(L)*\text{sub}(A)*\text{inv}(L^T)$; If $ibtype = 2$ or 3 , compute $U*\text{sub}(A)*U^T$, or $L^T*\text{sub}(A)*L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If $uplo = 'U'$, the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as U^T*U . If $uplo = 'L'$, the lower triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $L*L^T$.
<i>n</i>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pssygst DOUBLE PRECISION for pdsygst.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, the array contains the local pieces of the n -by- n symmetric distributed matrix $sub(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $sub(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $sub(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

b

(local)

REAL for pssygst

DOUBLE PRECISION for pdsygst.

Pointer into the local memory to an array of size $(lld_b, LOCC(jb+n-1))$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of $sub(B)$ as returned by $p?potrf$.

ib, jb

(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.

descb

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .

Output Parameters

a

On exit, if $info = 0$, the transformed matrix, stored in the same format as $sub(A)$.

scale

(global)

REAL for pssygst

DOUBLE PRECISION for pdsygst.

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, $scale$ is always returned as 1.0, it is returned here to allow for future enhancement.

info

(global) INTEGER.

If $info = 0$, the execution is successful. If $info < 0$, if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?hegst

Reduces a Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
call pchegst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

```
call pzhegst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

Include Files

Description

The p?hegst routine reduces complex Hermitian positive-definite generalized eigenproblems to the standard form.

In the following sub(A) denotes $A(ia:ia+n-1, ja:ja+n-1)$ and sub(B) denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If *ibtype* = 1, the problem is

$$\text{sub}(A)*x = \lambda*\text{sub}(B)*x,$$

and sub(A) is overwritten by $\text{inv}(U^H)*\text{sub}(A)*\text{inv}(U)$, or $\text{inv}(L)*\text{sub}(A)*\text{inv}(L^H)$.

If *ibtype* = 2 or 3, the problem is

$$\text{sub}(A)*\text{sub}(B)*x = \lambda*x, \text{ or } \text{sub}(B)*\text{sub}(A)*x = \lambda*x,$$

and sub(A) is overwritten by $U*\text{sub}(A)*U^H$, or $L^H*\text{sub}(A)*L$.

sub(B) must have been previously factorized as U^H*U or $L*L^H$ by p?potrf.

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, compute $\text{inv}(U^H)*\text{sub}(A)*\text{inv}(U)$, or $\text{inv}(L)*\text{sub}(A)*\text{inv}(L^H)$; If <i>itype</i> = 2 or 3, compute $U*\text{sub}(A)*U^H$, or $L^H*\text{sub}(A)*L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of sub(A) is stored and sub (B) is factored as U^H*U . If <i>uplo</i> = 'L', the lower triangle of sub(A) is stored and sub (B) is factored as $L*L^H$.
<i>n</i>	(global) INTEGER. The order of the matrices sub (A) and sub (B) ($n \geq 0$).
<i>a</i>	(local) COMPLEX for pchegst DOUBLE COMPLEX for pzhegst. Pointer into the local memory to an array of size $(l1d_a, LOCC(ja+n-1))$. On entry, the array contains the local pieces of the <i>n</i> -by- <i>n</i> Hermitian distributed matrix sub(A). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(A) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

<i>ia, ja</i>	(global) <code>INTEGER</code> . The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) <code>INTEGER</code> array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) <code>COMPLEX</code> for <code>pchegst</code> <code>DOUBLE COMPLEX</code> for <code>pzhgst</code> . Pointer into the local memory to an array of size $(lld_b, LOCC(jb+n-1))$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (<i>B</i>) as returned by <code>p?potrf</code> .
<i>ib, jb</i>	(global) <code>INTEGER</code> . The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) <code>INTEGER</code> array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) <code>REAL</code> for <code>pchegst</code> <code>DOUBLE PRECISION</code> for <code>pzhgst</code> . Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global) <code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Driver Routines

Table "ScaLAPACK Driver Routines" lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	<code>p?gesv</code> (simple driver) / <code>p?gesvx</code> (expert driver)

Type of Problem	Matrix type, storage scheme	Driver
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver) / p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general m -by- n	p?gels
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev / p?heev (simple driver); p?syevd / p?heevd (simple driver with a divide and conquer algorithm); p?syevx / p?heevx (expert driver); p?syevr / p?heevr (simple driver with MRRR algorithm)
Singular value decomposition	general m -by- n	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

[p?gesv](#)

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
call psgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

Include Files

Description

The `p?gesv` routine computes the solution to a real or complex system of linear equations $\text{sub}(A) * X = \text{sub}(B)$, where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is an n -by- n distributed matrix and X and $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$ are n -by- $nrhs$ distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. L and U are stored in $\text{sub}(A)$. The factored form of $\text{sub}(A)$ is then used to solve the system of equations $\text{sub}(A) * X = \text{sub}(B)$.

Input Parameters

n (global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

$nrhs$ (global) INTEGER. The number of right hand sides, that is, the number of columns of the distributed submatrices B and X ($nrhs \geq 0$).

a, b (local)

REAL for psgesv
 DOUBLE PRECISION for pdgesv
 COMPLEX for pcgesv
 DOUBLE COMPLEX for pzgesv.

Pointers into the local memory to arrays of local size $a(lld_a, LOCc(ja+n-1))$ and $b(lld_b, LOCc(jb+nrhs-1))$, respectively.

On entry, the array a contains the local pieces of the n -by- n distributed matrix $sub(A)$ to be factored.

On entry, the array b contains the right hand side distributed matrix $sub(B)$.

ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of $sub(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
ib, jb	(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of $sub(B)$, respectively.
$descb$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .

Output Parameters

a	Overwritten by the factors L and U from the factorization $sub(A) = P*L*U$; the unit diagonal elements of L are not stored .
b	Overwritten by the solution distributed matrix X .
$ipiv$	(local) INTEGER Array of size $LOCr(m_a)+mb_a$. This array contains the pivoting information. The (local) row i of the matrix was interchanged with the (global) row $ipiv(i)$. This array is tied to the distributed matrix A .
$info$	(global) INTEGER. If $info=0$, the execution is successful. $info < 0$: If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$. $info > 0$: If $info = k$, $U(ia+k-1, ja+k-1)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gesvx

Uses the *LU* factorization to compute the solution to the system of linear equations with a square matrix *A* and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
call psgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
iwork, liwork, info)
```

```
call pdgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
iwork, liwork, info)
```

```
call pcgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
rwork, lrwork, info)
```

```
call pzgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
rwork, lrwork, info)
```

Include Files

Description

The `p?gesvx` routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, where *A* denotes the *n*-by-*n* submatrix $A(ia:ia+n-1, ja:ja+n-1)$, *B* denotes the *n*-by-*nrhs* submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$ and *X* denotes the *n*-by-*nrhs* submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$.

Error bounds on the solution and a condition estimate are also provided.

In the following description, *af* stands for the subarray $af(iaf:iaf+n-1, jaf:jaf+n-1)$.

The routine `p?gesvx` performs the following steps:

1. If *fact* = 'E', real scaling factors *R* and *C* are computed to equilibrate the system:

$$trans = 'N': \text{diag}(R) * A * \text{diag}(C) * \text{diag}(C)^{-1} * X = \text{diag}(R) * B$$

$$trans = 'T': (\text{diag}(R) * A * \text{diag}(C))^T * \text{diag}(R)^{-1} * X = \text{diag}(C) * B$$

$$trans = 'C': (\text{diag}(R) * A * \text{diag}(C))^H * \text{diag}(R)^{-1} * X = \text{diag}(C) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by $\text{diag}(R) * A * \text{diag}(C)$ and *B* by $\text{diag}(R) * B$ (if *trans* = 'N') or $\text{diag}(C) * B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as $A = PLU$, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by $\text{diag}(C)$ (if *trans* = 'N') or $\text{diag}(R)$ (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>(global) CHARACTER*1. Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F' then, on entry, <i>af</i> and <i>ipiv</i> contain the factored form of <i>A</i>. If <i>equed</i> is not 'N', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>r</i> and <i>c</i>. Arrays <i>a</i>, <i>af</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix <i>A</i> is copied to <i>af</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix <i>A</i> is equilibrated if necessary, then copied to <i>af</i> and factored.</p>
<i>trans</i>	<p>(global) CHARACTER*1. Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A*X = B$ (No transpose);</p> <p>If <i>trans</i> = 'T', the system has the form $A^T*X = B$ (Transpose);</p> <p>If <i>trans</i> = 'C', the system has the form $A^H*X = B$ (Conjugate transpose);</p>
<i>n</i>	<p>(global) INTEGER. The number of linear equations; the order of the submatrix <i>A</i> ($n \geq 0$).</p>
<i>nrhs</i>	<p>(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices <i>B</i> and <i>X</i> ($nrhs \geq 0$).</p>
<i>a, af, b, work</i>	<p>(local)</p> <p>REAL for psgesvx DOUBLE PRECISION for pdgesvx COMPLEX for pcgesvx DOUBLE COMPLEX for pzgesvx.</p> <p>Pointers into the local memory to arrays of local size $a(lld_a, LOCC(ja+n-1))$, $af(lld_af, LOCC(ja+n-1))$, $b(lld_b, LOCC(jb+nrhs-1))$, $work(lwork)$.</p> <p>The array <i>a</i> contains the matrix <i>A</i>. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>A</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i>.</p> <p>The array <i>af</i> is an input argument if <i>fact</i> = 'F'. In this case it contains on entry the factored form of the matrix <i>A</i>, that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P*L*U$ as computed by <code>p?getrf</code>. If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix <i>A</i>.</p> <p>The array <i>b</i> contains on entry the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p> <p><i>work</i> is a workspace array. The size of <i>work</i> is (<i>lwork</i>).</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.</p>

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global matrix <i>AF</i> indicating the first row and the first column of the subarray <i>af(iaf:iaf+n-1, jaf:jaf+n-1)</i> , respectively.
<i>descaf</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B(ib:ib+n-1, jb:jb+nrhs-1)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ipiv</i>	(local) INTEGER Array of size $LOCr(m_a)+mb_a$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F' . On entry, it contains the pivot indices from the factorization $A = P*L*U$ as computed by <code>p?getrf</code> ; (local) row <i>i</i> of the matrix was interchanged with the (global) row <i>ipiv(i)</i> . This array must be aligned with $A(ia:ia+n-1, *)$.
<i>equed</i>	(global) CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. <i>equed</i> is an input argument if <i>fact</i> = 'F' . It specifies the form of equilibration that was done: If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N'); If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by $diag(r)$; If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by $diag(c)$; If <i>equed</i> = 'B', both row and column equilibration was done; <i>A</i> has been replaced by $diag(r)*A*diag(c)$.
<i>r, c</i>	(local) REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays of size $LOCr(m_a)$ and $LOCc(n_a)$, respectively. The array <i>r</i> contains the row scale factors for <i>A</i> , and the array <i>c</i> contains the column scale factors for <i>A</i> . These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments. If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by $diag(r)$; if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive. If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by $diag(c)$; if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.

If $fact = 'F'$ and $equed = 'C'$ or $'B'$, each element of c must be positive. Array r is replicated in every process column, and is aligned with the distributed matrix A . Array c is replicated in every process row, and is aligned with the distributed matrix A .

ix, jx	(global) INTEGER. The row and column indices in the global matrix X indicating the first row and the first column of the submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$, respectively.
$descx$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix X .
$lwork$	(local or global) INTEGER. The size of the array $work$; must be at least $\max(p?gecon(lwork), p?gerfs(lwork))+LOCr(n_a)$.
$iwork$	(local, psgesvx/pdgesvx only) INTEGER. Workspace array. The size of $iwork$ is ($liwork$).
$liwork$	(local, psgesvx/pdgesvx only) INTEGER. The size of the array $iwork$, must be at least $LOCr(n_a)$.
$rwork$	(local) REAL for pcgesvx DOUBLE PRECISION for pzgesvx. Workspace array, used in complex flavors only. The size of $rwork$ is ($lrwork$).
$lrwork$	(local or global, pcgesvx/pzgesvx only) INTEGER. The size of the array $rwork$; must be at least $2*LOCc(n_a)$.

Output Parameters

x	(local) REAL for psgesvx DOUBLE PRECISION for pdgesvx COMPLEX for pcgesvx DOUBLE COMPLEX for pzgesvx. Pointer into the local memory to an array of local size $x(lld_x, LOCc(jx+nrhs-1))$. If $info = 0$, the array x contains the solution matrix X to the <i>original</i> system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the <i>equilibrated</i> system is: $diag(C)^{-1} * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$; and $diag(R)^{-1} * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$.
a	Array a is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$. If $equed \neq 'N'$, A is scaled on exit as follows: $equed = 'R'$: $A = diag(R) * A$

	<code>equed = 'C': A = A*diag(c)</code>
	<code>equed = 'B': A = diag(R)*A*diag(c)</code>
<code>af</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>af</code> is an output argument and on exit returns the factors L and U from the factorization $A = P*L*U$ of the original matrix A (if <code>fact = 'N'</code>) or of the equilibrated matrix A (if <code>fact = 'E'</code>). See the description of <code>a</code> for the form of the equilibrated matrix.
<code>b</code>	Overwritten by <code>diag(R)*B</code> if <code>trans = 'N'</code> and <code>equed = 'R'</code> or <code>'B'</code> ; overwritten by <code>diag(c)*B</code> if <code>trans = 'T'</code> and <code>equed = 'C'</code> or <code>'B'</code> ; not changed if <code>equed = 'N'</code> .
<code>r, c</code>	These arrays are output arguments if <code>fact≠'F'</code> . See the description of <code>r, c</code> in <i>Input Arguments</i> section.
<code>rcond</code>	(global)REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix A after equilibration (if done). The routine sets <code>rcond = 0</code> if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <code>rcond</code> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<code>ferr, berr</code>	(local)REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays of size <code>LOCc(n_b)</code> each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector. Arrays <code>ferr</code> and <code>berr</code> are both replicated in every process row, and are aligned with the matrices B and X .
<code>ipiv</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>ipiv</code> is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix A (if <code>fact = 'N'</code>) or of the equilibrated matrix A (if <code>fact = 'E'</code>).
<code>equed</code>	If <code>fact≠'F'</code> , then <code>equed</code> is an output argument. It specifies the form of equilibration that was done (see the description of <code>equed</code> in <i>Input Arguments</i> section).
<code>work(1)</code>	If <code>info=0</code> , on exit <code>work(1)</code> returns the minimum value of <code>lwork</code> required for optimum performance.
<code>iwork(1)</code>	If <code>info=0</code> , on exit <code>iwork(1)</code> returns the minimum value of <code>liwork</code> required for optimum performance.
<code>rwork(1)</code>	If <code>info=0</code> , on exit <code>rwork(1)</code> returns the minimum value of <code>lrwork</code> required for optimum performance.
<code>info</code>	INTEGER. If <code>info=0</code> , the execution is successful.

$info < 0$: if the i th argument is an array and the j th entry had an illegal value, then $info = -(i*100+j)$; if the i th argument is a scalar and had an illegal value, then $info = -i$. If $info = i$, and $i \leq n$, then $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

```
call psgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pdgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pcgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pzgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
```

Include Files

Description

The `p?gbsv` routine computes the solution to a real or complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real/complex general banded distributed matrix with bwl subdiagonals and bwu superdiagonals, and X and $\text{sub}(B) = B(ib:ib+n-1, 1:rhs)$ are n -by- $nrhs$ distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U * Q$, where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of subdiagonals within the band of <i>A</i> ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of superdiagonals within the band of <i>A</i> ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed matrix sub(<i>B</i>) ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for psgbsv DOUBLE PRECISION for pdgbsv COMPLEX for pcgbsv DOUBLE COMPLEX for pzgbsv. Pointers into the local memory to arrays of local size $a(lld_a, LOCC(ja + n - 1))$ and $b(lld_b, LOCC(nrhs))$. On entry, the array <i>a</i> contains the local pieces of the global array <i>A</i> . On entry, the array <i>b</i> contains the right hand side distributed matrix sub(<i>B</i>).
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen</i> . The array descriptor for the distributed matrix <i>A</i> . If $desca(dtype_)$ = 501, then $dlen \geq 7$; else if $desca(dtype_)$ = 1, then $dlen \geq 9$.
<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of size <i>dlen</i> . The array descriptor for the distributed matrix <i>B</i> . If $descb(dtype_)$ = 502, then $dlen \geq 7$; else if $descb(dtype_)$ = 1, then $dlen \geq 9$.
<i>work</i>	(local) REAL for psgbsv DOUBLE PRECISION for pdgbsv COMPLEX for pcgbsv DOUBLE COMPLEX for pzgbsv.

Workspace array of size *lwork*.

lwork (local or global) INTEGER. The size of the array *work*, must be at least $lwork \geq (NB+bwu) * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2 * bwu) + \max(nrhs * (NB+2 * bwl+4 * bwu), 1)$.

Output Parameters

a On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

ipiv (local) INTEGER array.
The size of *ipiv* must be at least *desca*(NB). This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
If the *i*th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
call psdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pddbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The `p?dbsv` routine solves the following system of linear equations:

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded diagonally dominant-like distributed matrix with bandwidth bwl , bwu .

Gaussian elimination without pivoting is used to factor a reordering of the matrix into LU .

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

n	(global) INTEGER. The order of the distributed submatrix A , ($n \geq 0$).
bwl	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
bwu	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
$nrhs$	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B , ($nrhs \geq 0$).
a	(local). REAL for <code>psdbsv</code> DOUBLE PRECISION for <code>pddbsv</code> COMPLEX for <code>pcdbsv</code> DOUBLE COMPLEX for <code>pzdbsv</code> . Pointer into the local memory to an array with leading size $lld_a \geq (bwl + bwu + 1)$ (stored in <code>desca</code>). On entry, this array contains the local pieces of the distributed matrix.
ja	(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
$desca$	(global and local) INTEGER array of size $dlen$. If $1d$ type ($dtype_a=501$ or 502), $dlen \geq 7$; If $2d$ type ($dtype_a=1$), $dlen \geq 9$. The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
b	(local)

REAL for psdbsv
 DOUBLE PRECISION for pddbsv
 COMPLEX for pcdbsv
 DOUBLE COMPLEX for pzdbsv.

Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

ib (global) INTEGER. The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).

descb (global and local) INTEGER array of size $dlen$.
 If 1d type ($dtype_b = 502$), $dlen \geq 7$;
 If 2d type ($dtype_b = 1$), $dlen \geq 9$.
 The array descriptor for the distributed matrix B .
 Contains information of mapping of B to memory.

work (local).
 REAL for psdbsv
 DOUBLE PRECISION for pddbsv
 COMPLEX for pcdbsv
 DOUBLE COMPLEX for pzdbsv.
 Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

$$lwork \geq nb(bwl + bwu) + 6 \max(bwl, bwu) * \max(bwl, bwu) + \max(\max(bwl, bwu) nrhs, \max(bwl, bwu) * \max(bwl, bwu))$$

Output Parameters

a On exit, this array contains information containing details of the factorization.

Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

b On exit, this contains the local piece of the solutions distributed matrix X .

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.
 < 0 : If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0 : If $info = k < NPROCS$, the submatrix stored on processor $info$ and factored locally was not positive definite, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info - NPROCS$ representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
call psdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pddtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pcdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pzdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into LU .

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

n (global) INTEGER. The order of the distributed submatrix A ($n \geq 0$).

$nrhs$ INTEGER. The number of right hand sides; the number of columns of the distributed matrix B ($nrhs \geq 0$).

dl (local).REAL for psdtsv
DOUBLE PRECISION for pddtsv

	<p>COMPLEX for pcdtsv</p> <p>DOUBLE COMPLEX for pzdtsv.</p> <p>Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, $dl(1)$ is not referenced, and dl must be aligned with d. Must be of size $> desca(nb_)$.</p>
<i>d</i>	<p>(local). REAL for psdtsv</p> <p>DOUBLE PRECISION for pddtsv</p> <p>COMPLEX for pcdtsv</p> <p>DOUBLE COMPLEX for pzdtsv.</p> <p>Pointer to local part of global vector storing the main diagonal of the matrix.</p>
<i>du</i>	<p>(local). REAL for psdtsv</p> <p>DOUBLE PRECISION for pddtsv</p> <p>COMPLEX for pcdtsv</p> <p>DOUBLE COMPLEX for pzdtsv.</p> <p>Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du(n)$ is not referenced, and du must be aligned with d.</p>
<i>ja</i>	<p>(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).</p>
<i>desca</i>	<p>(global and local) INTEGER array of size $dlen$.</p> <p>If 1d type ($dtype_a=501$ or 502), $dlen \geq 7$;</p> <p>If 2d type ($dtype_a=1$), $dlen \geq 9$.</p> <p>The array descriptor for the distributed matrix A. Contains information of mapping of A to memory.</p>
<i>b</i>	<p>(local)</p> <p>REAL for psdtsv</p> <p>DOUBLE PRECISION for pddtsv</p> <p>COMPLEX for pcdtsv</p> <p>DOUBLE COMPLEX for pzdtsv.</p> <p>Pointer into the local memory to an array of local lead size $lld_b > nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.</p>
<i>ib</i>	<p>(global) INTEGER. The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).</p>
<i>descb</i>	<p>(global and local) INTEGER array of size $dlen$.</p> <p>If 1d type ($dtype_b =502$), $dlen \geq 7$;</p> <p>If 2d type ($dtype_b =1$), $dlen \geq 9$.</p>

The array descriptor for the distributed matrix B .
Contains information of mapping of B to memory.

work

(local).

REAL for psdtsv

DOUBLE PRECISION for pddtsv

COMPLEX for pcdtsv

DOUBLE COMPLEX for pzdtsv. Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork

(local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. $lwork > (12 * NPCOL + 3 * nb) + \max((10 + 2 * \min(100, nrhs)) * NPCOL + 4 * nrhs, 8 * NPCOL)$

Output Parameters

dl

On exit, this array contains information containing the * factors of the matrix.

d

On exit, this array contains information containing the * factors of the matrix. Must be of size $> desca(nb_)$.

du

On exit, this array contains information containing the * factors of the matrix. Must be of size $> desca(nb_)$.

b

On exit, this contains the local piece of the solutions distributed matrix X .

work

On exit, *work*(1) contains the minimal *lwork*.

info

(local) INTEGER. If *info*=0, the execution is successful.

< 0 : If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0 : If $info = k < NPROCS$, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info - NPROCS$ representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

call psposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)

```
call pdposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

Include Files

Description

The `p?posv` routine computes the solution to a real/complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and is an n -by- n symmetric/Hermitian distributed positive definite matrix and X and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ distributed matrices. The Cholesky decomposition is used to factor $\text{sub}(A)$ as

$$\text{sub}(A) = U^T * U, \text{ if } uplo = 'U', \text{ or}$$

$$\text{sub}(A) = L * L^T, \text{ if } uplo = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of $\text{sub}(A)$ is then used to solve the system of equations.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of $\text{sub}(A)$ is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a</i>	(local) REAL for <code>psposv</code> DOUBLE PRECISION for <code>pdposv</code> COMPLEX for <code>pcposv</code> COMPLEX*16 for <code>pzposv</code> . Pointer into the local memory to an array of size $(l1d_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$ to be factored. If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for <i>psposv</i> DOUBLE PRECISION for <i>pdposv</i> COMPLEX for <i>pcposv</i> COMPLEX*16 for <i>pzposv</i> . Pointer into the local memory to an array of size $(lld_b, LOCC(jb + nrhs - 1))$. On entry, the local pieces of the right hand sides distributed matrix <i>sub(B)</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, this array contains the local pieces of the factor <i>U</i> or <i>L</i> from the Cholesky factorization $sub(A) = U^H * U$, or $L * L^H$.
<i>b</i>	On exit, if <i>info</i> = 0, <i>sub(B)</i> is overwritten by the solution distributed matrix <i>X</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i * 100 + j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. If <i>info</i> > 0: If <i>info</i> = <i>k</i> , the leading minor of order <i>k</i> , $A(ia:ia+k-1, ja:ja+k-1)$ is not positive definite, and the factorization could not be completed, and the solution has not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

```
call psposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

```
call pdposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

```
call pcposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

```
call pzposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

Include Files

Description

The `p?posvx` routine uses the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ to compute the solution to a real or complex system of linear equations

$$A(ia:ia+n-1, ja:ja+n-1)*X = B(ib:ib+n-1, jb:jb+nrhs-1),$$

where $A(ia:ia+n-1, ja:ja+n-1)$ is a n -by- n matrix and X and $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments y denotes $Y(iy:iy+m-1, jy:jy+k-1)$, an m -by- k matrix where y can be a , af , b and x .

The routine `p?posvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(sr)*A*\text{diag}(sc)*\text{inv}(\text{diag}(sc))*X = \text{diag}(sr)*B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(sr)*A*\text{diag}(sc)$ and B by $\text{diag}(sr)*B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$$A = U^T*U, \text{ if } uplo = 'U', \text{ or}$$

$$A = L*L^T, \text{ if } uplo = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(sr)$ so that it solves the original system before equilibration.

Input Parameters

`fact` (global) CHARACTER. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, af contains the factored form of A . If $equed = 'Y'$, the matrix A has been equilibrated with scaling factors given by s . a and af will not be modified.

If $fact = 'N'$, the matrix A will be copied to af and factored.

If $fact = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.

$uplo$	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored.
n	(global) INTEGER. The order of the distributed matrix $sub(A)$ ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices B and X . ($nrhs \geq 0$).
a	(local) REAL for <code>psposvx</code> DOUBLE PRECISION for <code>pdposvx</code> COMPLEX for <code>pcposvx</code> DOUBLE COMPLEX for <code>pzposvx</code> . Pointer into the local memory to an array of local size ($lld_a, LOCC(ja + n - 1)$). On entry, the symmetric/Hermitian matrix A , except if $fact = 'F'$ and $equed = 'Y'$, then A must contain the equilibrated matrix $diag(sr) * A * diag(sc)$. If $uplo = 'U'$, the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced. A is not modified if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$ on exit.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
af	(local) REAL for <code>psposvx</code> DOUBLE PRECISION for <code>pdposvx</code> COMPLEX for <code>pcposvx</code> DOUBLE COMPLEX for <code>pzposvx</code> . Pointer into the local memory to an array of local size ($lld_af, LOCC(ja + n - 1)$).

If $fact = 'F'$, then af is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$, in the same storage format as A . If $equed \neq 'N'$, then af is the factored form of the equilibrated matrix $diag(sr) * A * diag(sc)$.

iaf, jaf

(global) INTEGER. The row and column indices in the global matrix AF indicating the first row and the first column of the submatrix AF , respectively.

descaf

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix AF .

equed

(global) CHARACTER. Must be 'N' or 'Y'.

equed is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);

If $equed = 'Y'$, equilibration was done and A has been replaced by $diag(sr) * A * diag(sc)$.

sr

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdposvx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Array of size lld_a .

The array s contains the scale factors for A . This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $equed = 'N'$, s is not accessed.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

b

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdposvx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Pointer into the local memory to an array of local size $(lld_b, LOCC(jb + nrhs - 1))$. On entry, the n -by- $nrhs$ right-hand side matrix B .

ib, jb

(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.

descb

(global and local) INTEGER. Array of size $dlen_$. The array descriptor for the distributed matrix B .

x

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdposvx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Pointer into the local memory to an array of local size $(l1d_x, LOCc(jx + nrhs - 1))$.

ix, jx

(global) INTEGER. The row and column indices in the global matrix X indicating the first row and the first column of the submatrix X , respectively.

descx

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix X .

work

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdposvx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*. *lwork* is local input and must be at least $lwork = \max(p?pocon(lwork), p?porfs(lwork)) + LOCr(n_a)$.

$lwork = 3 * desca(l1d_)$.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

iwork

(local) INTEGER. Workspace array of size *liwork*.

liwork

(local or global)

INTEGER. The size of the array *iwork*. *liwork* is local input and must be at least $liwork = desca(l1d_) liwork = LOCr(n_a)$.

If $liwork = -1$, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit, if $fact = 'E'$ and $equed = 'Y'$, *a* is overwritten by $diag(sr) * a * diag(sc)$.

af

If $fact = 'N'$, then *af* is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the original matrix A .

If $fact = 'E'$, then af is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

equed If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

sr This array is an output argument if $fact \neq 'F'$.
See the description of *sr* in *Input Arguments* section.

sc This array is an output argument if $fact \neq 'F'$.
See the description of *sc* in *Input Arguments* section.

b On exit, if $equed = 'N'$, b is not modified; if $trans = 'N'$ and $equed = 'R'$ or $'B'$, b is overwritten by $diag(r) * b$; if $trans = 'T'$ or $'C'$ and $equed = 'C'$ or $'B'$, b is overwritten by $diag(c) * b$.

x (local)
REAL for psposvx
DOUBLE PRECISION for pdposvx
COMPLEX for pcposvx
DOUBLE COMPLEX for pzposvx.
If $info = 0$ the n -by- $nrhs$ solution matrix X to the original system of equations.
Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the equilibrated system is
 $inv(diag(sc)) * X$ if $trans = 'N'$ and $equed = 'C'$ or $'B'$, or
 $inv(diag(sr)) * X$ if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$.

rcond (global)
REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision (in particular, if $rcond=0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr REAL for single precision flavors.
DOUBLE PRECISION for double precision flavors.
Arrays of size at least $\max(LOC, n_b)$. The estimated forward error bounds for each solution vector $X(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution, $ferr(j)$ bounds the magnitude of the largest entry in $(X(j) - xtrue)$ divided by the magnitude of the largest entry in

$X(j)$. The quality of the error bound depends on the quality of the estimate of $\text{norm}(\text{inv}(A))$ computed in the code; if the estimate of $\text{norm}(\text{inv}(A))$ is accurate, the error bound is guaranteed.

berr

(local)

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays of size at least $\max(LOC, n_b)$. The componentwise relative backward error of each solution vector $X(j)$ (the smallest relative change in any entry of A or B that makes $X(j)$ an exact solution).

work(1)

(local) On exit, *work(1)* returns the minimal and optimal *liwork*.

info

(global) INTEGER.

If *info*=0, the execution is successful.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: if *info* = *i*, and *i* is $\leq n$: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.

= *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
call pspbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pcpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pzpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The p?pbsv routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded symmetric positive definite distributed matrix with bandwidth bw .

Cholesky factorization is used to factor a reordering of the matrix into $L * L'$.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. Indicates whether the upper or lower triangular of <i>A</i> is stored. If <i>uplo</i> = 'U', the upper triangular <i>A</i> is stored If <i>uplo</i> = 'L', the lower triangular of <i>A</i> is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix <i>A</i> ($n \geq 0$).
<i>bw</i>	(global) INTEGER. The number of subdiagonals in <i>L</i> or <i>U</i> . $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>a</i>	(local). REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Pointer into the local memory to an array with leading size $lld_a \geq (bw+1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the distributed matrix $sub(A)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of size <i>dlen</i> . If 1D type (<i>dtype_b</i> =502), <i>dlen</i> ≥ 7; If 2D type (<i>dtype_b</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for pspbsv DOUBLE PRECISION for pdpbsv COMPLEX for pcpbsv DOUBLE COMPLEX for pzpbsv. Temporary workspace. This space may be overwritten in between calls to routines. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned. $lwork \geq (nb+2*bw)*bw + \max(bw*nrhs, bw*bw)$

Output Parameters

<i>a</i>	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>b</i>	On exit, contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i>	On exit, <i>work</i> (1) contains the minimal <i>lwork</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: If <i>info</i> = <i>k</i> ≤ NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ptsv

Syntax

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

```
call psptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pdptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pcptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

```
call pzptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

Include Files

Description

The `p?ptsv` routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into $L * L'$.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

n	(global) INTEGER. The order of matrix A ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
d	(local) REAL for <code>psptsv</code> DOUBLE PRECISION for <code>pdptsv</code> COMPLEX for <code>pcptsv</code> DOUBLE COMPLEX for <code>pzptsv</code> . Pointer to local part of global vector storing the main diagonal of the matrix.
e	(local) REAL for <code>psptsv</code> DOUBLE PRECISION for <code>pdptsv</code> COMPLEX for <code>pcptsv</code> DOUBLE COMPLEX for <code>pzptsv</code> .

	Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du(n)$ is not referenced, and du must be aligned with d .
<i>ja</i>	(global) INTEGER. The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array of size $dlen$. If 1d type ($dtype_a=501$ or 502), $dlen \geq 7$; If 2d type ($dtype_a=1$), $dlen \geq 9$. The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
<i>b</i>	(local) REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).
<i>descb</i>	(global and local) INTEGER array of size $dlen$. If 1d type ($dtype_b = 502$), $dlen \geq 7$; If 2d type ($dtype_b = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix B . Contains information of mapping of B to memory.
<i>work</i>	(local). REAL for psptsv DOUBLE PRECISION for pdptsv COMPLEX for pcptsv DOUBLE COMPLEX for pzptsv. Temporary workspace. This space may be overwritten in between calls to routines. $work$ must be the size given in $lwork$.
<i>lwork</i>	(local or global) INTEGER. Size of user-input workspace $work$. If $lwork$ is too small, the minimal acceptable size will be returned in $work(1)$ and an error code is returned. $lwork > (12 * NPCOL + 3 * nb) + \max((10 + 2 * \min(100, nrhs)) * NPCOL + 4 * nrhs, 8 * NPCOL)$.

Output Parameters

<i>d</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca(nb_)</i> .
<i>e</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca(nb_)</i> .
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i>	On exit, <i>work(1)</i> contains the minimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. If <i>info</i> =0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = $-(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. > 0: If <i>info</i> = $k \leq \text{NPROCS}$, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = $k > \text{NPROCS}$, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

```
call psgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pdgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pcgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pzgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

Include Files

Description

The `p?gels` routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, or its transpose/ conjugate-transpose, using a QTQ or LQ factorization of $\text{sub}(A)$. It is assumed that $\text{sub}(A)$ has full rank.

The following options are provided:

1. If *trans* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } ||\text{sub}(B) - \text{sub}(A)*X||$$
2. If *trans* = 'N' and $m < n$: find the minimum norm solution of an underdetermined system $\text{sub}(A)*X = \text{sub}(B)$.

3. If $trans = 'T'$ and $m \geq n$: find the minimum norm solution of an undetermined system $sub(A)^T X = sub(B)$.
4. If $trans = 'T'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||sub(B) - sub(A)^T X||$,

where $sub(B)$ denotes $B(ib:ib+m-1, jb:jb+nrhs-1)$ when $trans = 'N'$ and $B(ib:ib+n-1, jb:jb+nrhs-1)$ otherwise. Several right hand side vectors b and solution vectors x can be handled in a single call; when $trans = 'N'$, the solution vectors are stored as the columns of the n -by- $nrhs$ right hand side matrix $sub(B)$ and the m -by- $nrhs$ right hand side matrix $sub(B)$ otherwise.

Input Parameters

<i>trans</i>	(global) CHARACTER. Must be 'N', or 'T'. If $trans = 'N'$, the linear system involves matrix $sub(A)$; If $trans = 'T'$, the linear system involves the transposed matrix A^T (for real flavors only).
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $sub(A)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $sub(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices $sub(B)$ and X . ($nrhs \geq 0$).
<i>a</i>	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, contains the m -by- n matrix A .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>b</i>	(local) REAL for psgels DOUBLE PRECISION for pdgels COMPLEX for pcgels DOUBLE COMPLEX for pzgels.

Pointer into the local memory to an array of local size ($lld_b, LOCC(jb + nrhs - 1)$). On entry, this array contains the local pieces of the distributed matrix B of right-hand side vectors, stored columnwise; $sub(B)$ is m -by- $nrhs$ if $trans='N'$, and n -by- $nrhs$ otherwise.

ib, jb

(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.

descb

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .

work

(local)

REAL for psgels

DOUBLE PRECISION for pdgels

COMPLEX for pcgels

DOUBLE COMPLEX for pzgels.

Workspace array with size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work/lwork* is local input and must be at least $lwork \geq ltau + \max(lwf, lws)$, where if $m > n$, then

```
ltau = numroc(ja+min(m,n)-1, nb_a, MYCOL, csrc_a, NPCOL),
```

```
lwf = nb_a*(mpa0 + nqa0 + nb_a)
```

```
lws = max((nb_a*(nb_a-1))/2, (nrhsqb0 + mpb0)*nb_a) +
nb_a*nb_a
```

else

```
ltau = numroc(ia+min(m,n)-1, mb_a, MYROW, rsrc_a, NPROW),
```

```
lwf = mb_a * (mpa0 + nqa0 + mb_a)
```

```
lws = max((mb_a*(mb_a-1))/2, (npb0 + max(nqa0 +
numroc(numroc(n+iroffa, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nrhsqb0))*mb_a) + mb_a*mb_a
```

end if,

where $lcmp = lcm/NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,

```
iroffa = mod(ia-1, mb_a),
```

```
icoffa = mod(ja-1, nb_a),
```

```
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
```

```
iacol = indxg2p(ja, nb_a, MYROW, rsrc_a, NPROW)
```

```
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
```

```
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
```

```
iroffb = mod(ib-1, mb_b),
```

```
icoffb = mod(jb-1, nb_b),
```

```
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
```

```

ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
mpb0 = numroc(m+iroffb, mb_b, MYROW, icrow, NPROW),
nqb0 = numroc(n+icoffb, nb_b, MYCOL, ibcol, NPCOL),

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW`, and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, If $m \geq n$, <code>sub(A)</code> is overwritten by the details of its QR factorization as returned by <code>p?geqrf</code> ; if $m < n$, <code>sub(A)</code> is overwritten by details of its LQ factorization as returned by <code>p?gelqf</code> .
<code>b</code>	On exit, <code>sub(B)</code> is overwritten by the solution vectors, stored columnwise: if <code>trans = 'N'</code> and $m \geq n$, rows 1 to n of <code>sub(B)</code> contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $n+1$ to m in that column; If <code>trans = 'N'</code> and $m < n$, rows 1 to n of <code>sub(B)</code> contain the minimum norm solution vectors; If <code>trans = 'T'</code> and $m \geq n$, rows 1 to m of <code>sub(B)</code> contain the minimum norm solution vectors; if <code>trans = 'T'</code> and $m < n$, rows 1 to m of <code>sub(B)</code> contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $m+1$ to n in that column.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info = - (i* 100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
call pssyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
```

```
call pdsyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
```

Include Files

Description

The `p?syev` routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK routines.

In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) CHARACTER. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).
<i>a</i>	(local) REAL for <code>pssyev</code> . DOUBLE PRECISION for <code>pdsyev</code> . Block cyclic array of global size (n, n) and local size $(lld_a, LOCC(ja + n - 1))$. On entry, the symmetric matrix A . If <i>uplo</i> = 'U', only the upper triangular part of A is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of A is used to define the elements of the symmetric matrix.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iz, jz (global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*.

work (local)
 REAL for *pssyev*.
 DOUBLE PRECISION for *pdsyev*.
 Array of size *lwork*.

lwork (local) INTEGER. See below for definitions of variables used to define *lwork*.
 If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*n + sizesytrd + 1$,
 where *sizesytrd* is the workspace for *p?sytrd* and is $\max(NB*(np + 1), 3*NB)$.
 If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is:
 $qrmem = 2*n-2$
 $lwmin = 5*n + n*ldc + \max(sizemqrleft, qrmem) + 1$
 Variable definitions:
 $nb = desca(mb_) = desca(nb_) = descz(mb_) = descz(nb_);$
 $nn = \max(n, nb, 2);$
 $desca(rsrc_) = desca(rsrc_) = descz(rsrc_) = descz(csrc_) = 0$
 $np = \text{numroc}(nn, nb, 0, 0, NPROW)$
 $nq = \text{numroc}(\max(n, nb, 2), nb, 0, 0, NPCOL)$
 $nrc = \text{numroc}(n, nb, myprowc, 0, NPROCS)$
 $ldc = \max(1, nrc)$
sizemqrleft is the workspace for *p?ormtr* when its *side* argument is 'L'.
myprowc is defined when a new context is created as follows:
 call *blacs_get*(*desca(ctxt_)*, 0, *contextc*)
 call *blacs_gridinit*(*contextc*, 'R', NPROCS, 1)
 call *blacs_gridinfo*(*contextc*, *nprowc*, *npcolc*, *myprowc*, *mypcolc*)
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> ='L') or the upper triangle (if <i>uplo</i> ='U') of <i>A</i> , including the diagonal, is destroyed.
<i>w</i>	(global). REAL for <i>pssyev</i> DOUBLE PRECISION for <i>pdsyev</i> Array of size <i>n</i> . On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.
<i>z</i>	(local). REAL for <i>pssyev</i> DOUBLE PRECISION for <i>pdsyev</i> Array, global size (<i>n, n</i>), local size (<i>lld_z, LOcc(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work(1)</i>	On output, <i>work(1)</i> returns the workspace needed to guarantee completion. If the input parameters are incorrect, <i>work(1)</i> may also be incorrect. If <i>jobz</i> = 'N' <i>work(1)</i> = minimal (optimal) amount of workspace If <i>jobz</i> = 'V' <i>work(1)</i> = minimal workspace required to generate all the eigenvectors.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . If <i>info</i> > 0: If <i>info</i> = 1 through <i>n</i> , the <i>i</i> -th eigenvalue did not converge in <i>?steqr2</i> after a total of 30 <i>n</i> iterations. If <i>info</i> = <i>n</i> +1, then <i>p?syevev</i> has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from <i>p?syevev</i> cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevev

Computes all eigenvalues and eigenvectors of a real symmetric matrix by using a divide and conquer algorithm.

Syntax

```
call pssyevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, iwork, liwork, info)
```

call pdsyevd(*jobz*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *w*, *z*, *iz*, *jz*, *descz*, *work*, *lwork*, *iwork*, *liwork*, *info*)

Include Files

Description

The `p?syevd` routine computes all eigenvalues and eigenvectors of a real symmetric matrix A by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).
<i>a</i>	(local). REAL for pssyevd DOUBLE PRECISION for pdsyevd. Block cyclic array of global size (n, n) and local size $(lld_a, LOCC(ja + n - 1))$. On entry, the symmetric matrix A . If <i>uplo</i> = 'U', only the upper triangular part of A is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of A is used to define the elements of the symmetric matrix.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A . If <i>desca</i> (<i>ctxt_</i>) is incorrect, <code>p?syevd</code> cannot guarantee correct error reporting.
<i>iz</i> , <i>jz</i>	(global) INTEGER. The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i> .
<i>work</i>	(local). REAL for <i>pssyevd</i> DOUBLE PRECISION for <i>pdsyevd</i> . Array of size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The size of the array <i>work</i> . If eigenvalues are requested: $lwork \geq \max(1+6*n + 2*np*nq, trilwmin) + 2*n$ with $trilwmin = 3*n + \max(nb*(np + 1), 3*nb)$ $np = \text{numroc}(n, nb, myrow, iarow, NPROW)$ $nq = \text{numroc}(n, nb, mycol, iacol, NPCOL)$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by <i>pxerbla</i> .
<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> .
<i>liwork</i>	(local) INTEGER, size of <i>iwork</i> . $liwork = 7*n + 8*npcol + 2$.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L'), or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	(global). REAL for <i>pssyevd</i> DOUBLE PRECISION for <i>pdsyevd</i> . Array of size <i>n</i> . If <i>info</i> = 0, <i>w</i> contains the eigenvalues in the ascending order.
<i>z</i>	(local). REAL for <i>pssyevd</i> DOUBLE PRECISION for <i>pdsyevd</i> . Array, global size (<i>n, n</i>), local size (<i>lld_z, LOcc(jz+n-1)</i>). The <i>z</i> parameter contains the orthonormal eigenvectors of the matrix <i>A</i> .
<i>work</i> (1)	On exit, returns adequate workspace to allow optimal performance.
<i>iwork</i> (1)	(local). On exit, if <i>liwork</i> > 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$. If the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

The algorithm failed to compute the *info*/(*n*+1)-th eigenvalue while working on the submatrix lying in global rows and columns $\text{mod}(\text{info}, n+1)$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using Relatively Robust Representation.

Syntax

```
call pssyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
  iz, jz, descz, work, lwork, iwork, liwork, info )
```

```
call pdsyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
  iz, jz, descz, work, lwork, iwork, liwork, info )
```

Include Files**Description**

`p?syevr` computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK routines.

First, the matrix *A* is reduced to real symmetric tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in *w*. The eigenvector matrix *z* is stored in 2D block-cyclic format distributed over all processors.

Note that subsets of eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>jobz</i>	(global) CHARACTER*1 Specifies whether or not to compute the eigenvectors: = 'N': Compute eigenvalues only. = 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	(global) CHARACTER*1 = 'A': all eigenvalues will be found. = 'V': all eigenvalues in the interval [<i>vl</i> , <i>vu</i>] will be found. = 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>uplo</i>	(global) CHARACTER*1 Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	(global) INTEGER The number of rows and columns of the matrix <i>a</i> . $n \geq 0$
<i>a</i>	REAL for <i>pssyevr</i> DOUBLE PRECISION for <i>pdsyevr</i> Block cyclic array of global size (<i>n</i> , <i>n</i>), local size (<i>lld_a</i> , $LOC_c(ja+n-1)$). This array contains the local pieces of the symmetric distributed matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>a</i> is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>a</i> is used to define the elements of the symmetric matrix. On exit, the lower triangle (if <i>uplo</i> ='L') or the upper triangle (if <i>uplo</i> ='U') of <i>a</i> , including the diagonal, is destroyed.
<i>ia</i>	(global) INTEGER Global row index in the global matrix <i>A</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>ja</i>	(global) INTEGER Global column index in the global matrix <i>A</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> =9. The array descriptor for the distributed matrix <i>a</i> .
<i>vl</i>	REAL for <i>pssyevr</i> DOUBLE PRECISION for <i>pdsyevr</i> (global) If <i>range</i> ='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i>vu</i>	REAL for <i>pssyevr</i> DOUBLE PRECISION for <i>pdsyevr</i> (global) If <i>range</i> ='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i>	(global) INTEGER If <i>range</i> ='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. $il \geq 1$. Not referenced if <i>range</i> = 'A'.
<i>iu</i>	(global) INTEGER If <i>range</i> ='I', the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(il, n) \leq iu \leq n$. Not referenced if <i>range</i> = 'A'.
<i>iz</i>	(global) INTEGER Global row index in the global matrix <i>Z</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>jz</i>	(global) INTEGER Global column index in the global matrix <i>Z</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>descz</i>	INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>z</i> . The context <i>descz</i> (<i>ctxt_</i>) must equal <i>desca</i> (<i>ctxt_</i>). Also note the array alignment requirements specified below.
<i>work</i>	REAL for <i>pssyevr</i> DOUBLE PRECISION for <i>pdsyevr</i> (local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) INTEGER Size of <i>work</i> , must be at least 3.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N') then

$$lwork \geq 2 + 5*n + \max(12 * nn, neig * (np0 + 1))$$

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

$$lwork \geq 2 + 5*n + \max(18*nn, np0 * mq0 + 2 * neig * neig) + (2 + \text{iceil}(neig, nprow*npcol)) * nn$$

Variable definitions:

neig = number of eigenvectors requested

nb = *desca*(*mb_*) = *desca*(*nb_*) = *descz*(*mb_*) = *descz*(*nb_*)

nn = $\max(n, neig, 2)$

desca(*rsrc_*) = *desca*(*csrc_nb_*) = *descz*(*rsrc_*) = *descz*(*csrc_*) = 0

np0 = *numroc*(*nn*, *neig*, 0, 0, *nprow*)

mq0 = *numroc*($\max(neig, neig, 2)$, *neig*, 0, 0, *npcol*)

iceil(*x*, *y*) is a ScaLAPACK function returning $\text{ceiling}(x/y)$, and *nprow* and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by [pxerbla](#).

liwork

(local) INTEGER

size of *iwork*

Let *nnp* = $\max(n, nprow*npcol + 1, 4)$. Then:

liwork $\geq 12*nnp + 2*n$ when the eigenvectors are desired

liwork $\geq 10*nnp + 2*n$ when only the eigenvalues have to be computed

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

OUTPUT Parameters

m

(global) INTEGER

Total number of eigenvalues found. $0 \leq m \leq n$.

nz

(global) INTEGER

Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m*

<i>w</i>	REAL for pssyevr DOUBLE PRECISION for pdsyevr (global) array of size n Upon successful exit, the first m entries contain the selected eigenvalues in ascending order.
<i>z</i>	REAL for pssyevr DOUBLE PRECISION for pdsyevr Block-cyclic array, global size(n, n), local size ($lld_z, LOC_c(jz+n-1)$) . On exit, contains local pieces of distributed matrix Z .
<i>work</i>	On return, $work(1)$ contains the optimal amount of workspace required for efficient execution. If $jobz='N'$ $work(1)$ = optimal amount of workspace required to compute the eigenvalues. If $jobz='V'$ $work(1)$ = optimal amount of workspace required to compute eigenvalues and eigenvectors.
<i>iwork</i>	(local workspace) INTEGER array On return, $iwork(1)$ contains the amount of integer workspace required.
<i>info</i>	(global) INTEGER = 0: successful exit < 0: If the i -th argument is an array and the j th-entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The distributed submatrices $a(ia:*, ja:*)$ and $z(iz:iz+m-1, jz:jz+n-1)$ must satisfy the following alignment properties:

1. Identical (quadratic) dimension: $desca(m_) = descz(m_) = desca(n_) = descz(n_)$
2. Quadratic conformal blocking: $desca(mb_) = desca(nb_) = descz(mb_) = descz(nb_)$, $desca(rsrc_) = descz(rsrc_)$
3. $\text{mod}(ia-1, mb_a) = \text{mod}(iz-1, mb_z) = 0$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

call pssyevx(*jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info*)

call `pdsyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)`

Include Files

Description

The `p?syevx` routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	(global) CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval $[vl, vu]$ will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).
<i>a</i>	(local). REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Block cyclic array of global size (n, n) and local size $(lld_a, LOCC(ja + n - 1))$. On entry, the symmetric matrix A .

	<p>If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the symmetric matrix.</p> <p>If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the symmetric matrix.</p>
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
vl, vu	(global) REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if $range = 'A'$ or $'I'$.
il, iu	(global) INTEGER. If $range = 'I'$, the indices of the smallest and largest eigenvalues to be returned. Constraints: $il \geq 1$ $\min(il, n) \leq iu \leq n$ Not referenced if $range = 'A'$ or $'V'$.
$abstol$	(global).REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . If $jobz = 'V'$, setting $abstol$ to <code>p?lamch(context, 'U')</code> yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b)$, where eps is the machine precision. If $abstol$ is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $(\text{mod}(info, 2) \neq 0)$ or $(\text{mod}(info/8, 2) \neq 0)$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2 * p?lamch('S')$.
$orfac$	(global).REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient

(see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local)

REAL for *pssyevx*.

DOUBLE PRECISION for *pdsyevx*.

Array of size *lwork*.

lwork

(local) INTEGER. The size of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*n + \max(5*nn, NB*(np0 + 1))$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*NB*NB) + \text{iceil}(neig, NPROW*NPCOL)*nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid w(j+1) \leq w(j)\} + orfac*2*norm(A),$$

where

neig = number of eigenvectors requested

nb = *desca(mb_)* = *desca(nb_)* = *descz(mb_)* = *descz(nb_)*;

nn = $\max(n, nb, 2)$;

desca(rsrc_) = *desca(nb_)* = *descz(rsrc_)* = *descz(csrc_)* = 0;

np0 = *numroc(nn, nb, 0, 0, NPROW)*;

mq0 = *numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)*

iceil(x, y) is a ScaLAPACK function returning ceiling(*x/y*)

If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible.

Relationship between workspace, orthogonality & performance:

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

$$lwork \geq \max(lwork, 5*n + nsytrd_lwopt),$$

where *lwork*, as defined previously, depends upon the number of eigenvectors requested, and

$$nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps + 3)*nps;$$

$$anb = pjlaenv(desca(ctxt_), 3, 'p?sytrd', 'L', 0, 0, 0, 0);$$

$$sqnpc = \text{int}(\text{sqrt}(\text{dble}(\text{NPROW} * \text{NPCOL})));$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2*anb);$$

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function

MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a megabyte per process).

If *clustersize* > $n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*-1) *p?stein* will perform no better than *?stein* on single processor.

For *clustersize* = $n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* > $n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *p?erbla*.

iwork

(local) INTEGER. Workspace array.

liwork

(local) INTEGER, size of *iwork*. $liwork \geq 6*nnp$

Where: $nnp = \max(n, \text{NPROW}*\text{NPCOL} + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pserbla`.

Output Parameters

<code>a</code>	On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.
<code>m</code>	(global) INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.
<code>nz</code>	(global) INTEGER. Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of z that are filled. If $jobz \neq 'V'$, nz is not referenced. If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and <code>p?syevx</code> is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z ($m \leq descz(n_)$) and sufficient workspace to compute them. (See $lwork$). <code>p?syevx</code> is always able to detect insufficient space without computation unless $range = 'V'$.
<code>w</code>	(global).REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Array of size n . The first m elements contain the selected eigenvalues in ascending order.
<code>z</code>	(local).REAL for <code>pssyevx</code> DOUBLE PRECISION for <code>pdsyevx</code> . Array, global size (n, n) , local size $(lld_z, LOCC(jz+n-1))$. If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <code>ifail</code> . If $jobz = 'N'$, then z is not referenced.
<code>work(1)</code>	On exit, returns workspace adequate workspace to allow optimal performance.
<code>iwork(1)</code>	On return, <code>iwork(1)</code> contains the amount of integer workspace required.
<code>ifail</code>	(global)INTEGER. Array of size n . If $jobz = 'V'$, then on normal exit, the first m elements of <code>ifail</code> are zero. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then <code>ifail</code> contains the indices of the eigenvectors that failed to converge. If $jobz = 'N'$, then <code>ifail</code> is not referenced.

iclustr (global) INTEGER. Array of size $(2 * \text{NPROW} * \text{NPCOL})$

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed $iclustr(2*i-1)$ to $iclustr(2*i)$, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr()* is a zero terminated array. $iclustr(2*k) \neq 0$ and $iclustr(2*k+1) = 0$ if and only if k is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap (global)

REAL for *pssyevx*

DOUBLE PRECISION for *pdsyevx*.

Array of size $\text{NPROW} * \text{NPCOL}$

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*th cluster may be as high as $(C*n) / gap(i)$ where C is a small constant.

info (global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then $info = -(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

If *info* > 0: if $(\text{mod}(info, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure $abstol=2.0*p?lamch('U')$.

If $(\text{mod}(info/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(info/4, 2) \neq 0)$, then space limit prevented *p?syevxf* from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(info/8, 2) \neq 0)$, then *p?stebz* failed to compute eigenvalues. Ensure $abstol=2.0*p?lamch('U')$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heev

Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix.

Syntax

```
call pcheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork,
lrwork, info)
```

```
call pzheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork,
lrwork, info)
```

Include Files

Description

The p?heev routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK routines. The routine assumes a homogeneous system and makes spot checks of the consistency of the eigenvalues across the different processes. A heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).
<i>a</i>	(local). COMPLEX for pcheev DOUBLE COMPLEX for pzheev. Block cyclic array of global size (n, n) and local size $(lld_a, LOcc(ja + n - 1))$. On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', only the upper triangular part of A is used to define the elements of the Hermitian matrix. If <i>uplo</i> = 'L', only the lower triangular part of A is used to define the elements of the Hermitian matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, <i>p?heev</i> cannot guarantee correct error reporting.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i> .
<i>work</i>	(local). COMPLEX for <i>pcheev</i> DOUBLE COMPLEX for <i>pzheev</i> . Array of size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The size of the array <i>work</i> . If only eigenvalues are requested (<i>jobz</i> = 'N'): $lwork \geq \max(nb * (np0 + 1), 3) + 3 * n$ If eigenvectors are requested (<i>jobz</i> = 'V'), then the amount of workspace required: $lwork \geq (np0 + nq0 + nb) * nb + 3 * n + n^2$ with $nb = desca(mb_) = desca(nb_) = nb = descz(mb_) = descz(nb_)$ $np0 = \text{numroc}(nn, nb, 0, 0, \text{NPROW})$. $nq0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by <i>pxerbla</i> .
<i>rwork</i>	(local). REAL for <i>pcheev</i> DOUBLE PRECISION for <i>pzheev</i> . Workspace array of size <i>lrwork</i> .
<i>lrwork</i>	(local) INTEGER. The size of the array <i>rwork</i> . See below for definitions of variables used to define <i>lrwork</i> . If no eigenvectors are requested (<i>jobz</i> = 'N'), then $lrwork \geq 2 * n$. If eigenvectors are requested (<i>jobz</i> = 'V'), then $lrwork \geq 2 * n + 2 * n - 2$.

If $lrwork = -1$, then $lrwork$ is global input and a workspace query is assumed; the routine only calculates the minimum size required for the $rwork$ array. The required workspace is returned as the first element of $rwork$, and no error message is issued by `pserbla`.

Output Parameters

a	On exit, the lower triangle (if $uplo = 'L'$), or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.
w	(global). REAL for <code>pcheev</code> DOUBLE PRECISION for <code>pzheev</code> . Array of size n . The first m elements contain the selected eigenvalues in ascending order.
z	(local). COMPLEX for <code>pcheev</code> DOUBLE COMPLEX for <code>pzheev</code> . Array, global size (n, n) , local size $(lld_z, LOCC(jz+n-1))$. If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If $jobz = 'N'$, then z is not referenced.
$work(1)$	On exit, returns adequate workspace to allow optimal performance. If $jobz = 'N'$, then $work(1) =$ minimal workspace only for eigenvalues. If $jobz = 'V'$, then $work(1) =$ minimal workspace required to generate all the eigenvectors.
$rwork(1)$	(local) COMPLEX for <code>pcheev</code> DOUBLE COMPLEX for <code>pzheev</code> . On output, $rwork(1)$ returns workspace required to guarantee completion.
$info$	(global) INTEGER. If $info = 0$, the execution is successful. If $info < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$. If the i -th argument is a scalar and had an illegal value, then $info = -i$. If $info > 0$: If $info = 1$ through n , the i -th eigenvalue did not converge in <code>?steqr2</code> after a total of $30*n$ iterations.

If $info = n+1$, then `p?heev` detected heterogeneity, and the accuracy of the results cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevd

Computes all eigenvalues and eigenvectors of a complex Hermitian matrix by using a divide and conquer algorithm.

Syntax

```
call pcheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

```
call pzheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

Include Files

Description

The `p?heevd` routine computes all eigenvalues and eigenvectors of a complex Hermitian matrix A by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<code>jobz</code>	(global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: If <code>uplo = 'U'</code> , a stores the upper triangular part of A . If <code>uplo = 'L'</code> , a stores the lower triangular part of A .
<code>n</code>	(global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).
<code>a</code>	(local). COMPLEX for <code>pcheevd</code> DOUBLE COMPLEX for <code>pzheevd</code> . Block cyclic array of global size (n, n) and local size $(lld_a, LOcc(ja + n - 1))$. On entry, the Hermitian matrix A . If <code>uplo = 'U'</code> , only the upper triangular part of A is used to define the elements of the Hermitian matrix.

If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, <i>p?heevd</i> cannot guarantee correct error reporting.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i> .
<i>work</i>	(local). COMPLEX for <i>pcheevd</i> DOUBLE COMPLEX for <i>pzheevd</i> . Array of size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The size of the array <i>work</i> . If eigenvalues are requested: $lwork = n + (nb0 + mq0 + nb) * nb$ with $np0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPROW})$ $mq0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by <i>pxerbla</i> .
<i>rwork</i>	(local). REAL for <i>pcheevd</i> DOUBLE PRECISION for <i>pzheevd</i> . Workspace array of size <i>lrwork</i> .
<i>lrwork</i>	(local) INTEGER. The size of the array <i>rwork</i> . $lrwork \geq 1 + 9 * n + 3 * np * nq,$ with $np = \text{numroc}(n, nb, myrow, iarow, \text{NPROW})$ $nq = \text{numroc}(n, nb, mycol, iacol, \text{NPCOL})$
<i>iwork</i>	(local) INTEGER. Workspace array of size <i>liwork</i> .
<i>liwork</i>	(local) INTEGER, size of <i>iwork</i> . $liwork = 7 * n + 8 * npc0l + 2.$

Output Parameters

<code>a</code>	On exit, the lower triangle (if <code>uplo = 'L'</code>), or the upper triangle (if <code>uplo = 'U'</code>) of A , including the diagonal, is overwritten.
<code>w</code>	(global). REAL for <code>pcheevd</code> DOUBLE PRECISION for <code>pzheevd</code> . Array of size n . If <code>info = 0</code> , w contains the eigenvalues in the ascending order.
<code>z</code>	(local). COMPLEX for <code>pcheevd</code> DOUBLE COMPLEX for <code>pzheevd</code> . Array, global size (n, n) , local size $(lld_z, LOCC(jz+n-1))$. The z parameter contains the orthonormal eigenvectors of the matrix A .
<code>work(1)</code>	On exit, returns adequate workspace to allow optimal performance.
<code>rwork(1)</code>	(local) COMPLEX for <code>pcheevd</code> DOUBLE COMPLEX for <code>pzheevd</code> . On output, <code>rwork(1)</code> returns workspace required to guarantee completion.
<code>iwork(1)</code>	(local). On return, <code>iwork(1)</code> contains the amount of integer workspace required.
<code>info</code>	(global) INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info < 0</code> : If the i -th argument is an array and the j -entry had an illegal value, then <code>info = -(i*100+j)</code> . If the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> . If <code>info > 0</code> : If <code>info = 1</code> through n , the i -th eigenvalue did not converge.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using Relatively Robust Representation.

Syntax

```
call pcheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
            iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
```

```
call pzheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
            iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
```

Include Files

Description

`p?heevr` computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK routines.

First, the matrix A is reduced to complex Hermitian tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in w . The eigenvector matrix Z is stored in 2D block-cyclic format distributed over all processors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>jobz</i>	(global) CHARACTER*1 Specifies whether or not to compute the eigenvectors: = 'N': Compute eigenvalues only. = 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	(global) CHARACTER*1 = 'A': all eigenvalues will be found. = 'V': all eigenvalues in the interval [<i>vl</i> , <i>vu</i>] will be found. = 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>uplo</i>	(global) CHARACTER*1 Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	(global) INTEGER The number of rows and columns of the matrix A . $n \geq 0$
<i>a</i>	COMPLEX for <code>pcheevr</code> COMPLEX*16 for <code>pzheevr</code>

	Block-cyclic array, global size (n, n) , local size $(lld_a, LOC_c(ja+n-1))$
	Contains the local pieces of the Hermitian distributed matrix A . If $uplo = 'U'$, only the upper triangular part of a is used to define the elements of the Hermitian matrix. If $uplo = 'L'$, only the lower triangular part of a is used to define the elements of the Hermitian matrix.
<i>ia</i>	(global) INTEGER Global row index in the global matrix A that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>ja</i>	(global) INTEGER Global column index in the global matrix A that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. (The ScaLAPACK descriptor length is $dlen_ = 9$.) The array descriptor for the distributed matrix a . The descriptor stores details about the 2D block-cyclic storage, see the notes below. If <i>desca</i> is incorrect, <code>p?heevr</code> cannot work correctly. Also note the array alignment requirements specified below
<i>vl</i>	REAL for <code>pcheevr</code> DOUBLE PRECISION for <code>pzheevr</code> (global) If $range='V'$, the lower bound of the interval to be searched for eigenvalues. Not referenced if $range = 'A'$ or $'I'$.
<i>vu</i>	REAL for <code>pcheevr</code> DOUBLE PRECISION for <code>pzheevr</code> (global) If $range='V'$, the upper bound of the interval to be searched for eigenvalues. Not referenced if $range = 'A'$ or $'I'$.
<i>il</i>	(global) INTEGER If $range='I'$, the index (from smallest to largest) of the smallest eigenvalue to be returned. $il \geq 1$. Not referenced if $range = 'A'$.
<i>iu</i>	(global) INTEGER If $range='I'$, the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(il, n) \leq iu \leq n$. Not referenced if $range = 'A'$.
<i>iz</i>	(global) INTEGER

Global row index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

jz (global) INTEGER

Global column index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix z . *descz*(*ctxt_*) must equal *desca*(*ctxt_*)

work COMPLEX for *pcheevr*

COMPLEX*16 for *pzheevr*

(local workspace) array of size *lwork*

lwork (local) INTEGER

Size of *work* array, must be at least 3.

If only eigenvalues are requested:

$$lwork \geq n + \max(nb * (np00 + 1), nb * 3)$$

If eigenvectors are requested:

$$lwork \geq n + (np00 + mq00 + nb) * nb$$

For definitions of *np00* and *mq00*, see *lwork*.

For optimal performance, greater workspace is needed, i.e.

$$lwork \geq \max(lwork, nhetrd_lwork)$$

Where *lwork* is as defined above, and

$$nhetrd_lwork = n + 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$$

$$ictxt = desca(ctxt_)$$

$$anb = pjlaenv(ictxt, 3, 'PCHETTRD', 'L', 0, 0, 0, 0)$$

$$sqnpc = \text{sqrt}(\text{real}(nprow * npc0l))$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 * anb)$$

If *lwork* = -1, then *lwork* is global input and a workspace query is

assumed; the routine only calculates the optimal size for all work arrays.

Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

rwork REAL for *pcheevr*

DOUBLE PRECISION for *pzheevr*

(local workspace) array of size *lrwork*

lrwork (local) INTEGER

Size of *rwork*, must be at least 3.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested ($jobz = 'N'$) then

$$lwork \geq 2 + 5 * n + \max(12 * n, nb * (np00 + 1))$$

If eigenvectors are requested ($jobz = 'V'$) then the amount of workspace required is:

$$lwork \geq 2 + 5 * n + \max(18 * n, np00 * mq00 + 2 * nb * nb) + (2 + \text{iceil}(neig, nprow * npcol)) * n$$

NOTE

$\text{iceil}(x, y)$ is the ceiling of x/y .

Variable definitions:

$neig$ = number of eigenvectors requested

$nb = \text{desca}(mb_) = \text{desca}(nb_) = \text{descz}(mb_) = \text{descz}(nb_)$

$nn = \max(n, nb, 2)$

$\text{desca}(rsrc_) = \text{desca}(csrc_) = \text{descz}(rsrc_) = \text{descz}(csrc_) = 0$

$np00 = \text{numroc}(nn, nb, 0, 0, nprow)$

$mq00 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, npcol)$

$\text{iceil}(x, y)$ is a ScaLAPACK function returning $\text{ceiling}(x/y)$, and $nprow$ and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`

$iwork$ (local workspace) INTEGER array of size $liwork$

$liwork$ (local) INTEGER
size of $iwork$

Let $nnp = \max(n, nprow * npcol + 1, 4)$. Then:

$liwork \geq 12 * nnp + 2 * n$ when the eigenvectors are desired

$liwork \geq 10 * nnp + 2 * n$ when only the eigenvalues have to be computed

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`

OUTPUT Parameters

a The lower triangle (if $uplo='L'$) or the upper triangle (if $uplo='U'$) of a , including the diagonal, is destroyed.

m (global) INTEGER

Total number of eigenvalues found. $0 \leq m \leq n$.

<i>nz</i>	<p>(global) INTEGER</p> <p>Total number of eigenvectors computed. $0 \leq nz \leq m$.</p> <p>The number of columns of <i>z</i> that are filled.</p> <p>If <i>jobz</i> ≠ 'V', <i>nz</i> is not referenced.</p> <p>If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i></p>
<i>w</i>	<p>REAL for <i>pcheevr</i></p> <p>DOUBLE PRECISION for <i>pzheevr</i></p> <p>(global) array of size <i>n</i></p> <p>On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.</p>
<i>z</i>	<p>COMPLEX for <i>pcheevr</i></p> <p>COMPLEX*16 for <i>pzheevr</i></p> <p>(local) array, global size (<i>n</i>, <i>n</i>), local size (<i>lld_z</i>, <i>LOC_c(jz+n-1)</i>)</p> <p>If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work</i>	<p><i>work</i>(1) returns workspace adequate workspace to allow optimal performance.</p>
<i>rwork</i>	<p>On return, <i>rwork</i>(1) contains the optimal amount of workspace required for efficient execution. if <i>jobz</i>='N' <i>rwork</i>(1) = optimal amount of workspace required to compute the eigenvalues. if <i>jobz</i>='V' <i>rwork</i>(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors.</p>
<i>iwork</i>	<p>On return, <i>iwork</i>(1) contains the amount of integer workspace required.</p>
<i>info</i>	<p>(global) INTEGER</p> <p>= 0: successful exit</p> <p>< 0: If the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>), if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p>

Application Notes

The distributed submatrices *a*(*ia*:* , *ja*:*) and *z*(*iz*:*iz*+*m*-1, *jz*:*jz*+*n*-1) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desca*(*m*_) = *descz*(*m*_) = *desca*(*n*_) = *descz*(*n*_)
2. Quadratic conformal blocking: *desca*(*mb*_) = *desca*(*nb*_) = *descz*(*mb*_) = *descz*(*nb*_), *desca*(*rsrc*_) = *descz*(*rsrc*_)
3. $\text{mod}(ia-1, mb_a) = \text{mod}(iz-1, mb_z) = 0$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
call pcheevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz,
w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr,
gap, info)
```

```
call pzheevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz,
w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr,
gap, info)
```

Include Files**Description**

The `p?heevx` routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'.
 Specifies if it is necessary to compute the eigenvectors:
 If *jobz* = 'N', then only eigenvalues are computed.
 If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range (global) CHARACTER*1. Must be 'A', 'V', or 'I'.
 If *range* = 'A', all eigenvalues will be found.

	<p>If <i>range</i> = 'V', all eigenvalues in the half-open interval [<i>v1</i>, <i>vu</i>] will be found.</p> <p>If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.</p>
<i>uplo</i>	<p>(global) CHARACTER*1. Must be 'U' or 'L'.</p> <p>Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored:</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i>.</p>
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<p>(local).</p> <p>COMPLEX for <i>pcheevx</i></p> <p>DOUBLE COMPLEX for <i>pzheevx</i>.</p> <p>Block cyclic array of global size (<i>n</i>, <i>n</i>) and local size (<i>lld_a</i>, <i>LOCc(ja + n - 1)</i>). On entry, the Hermitian matrix <i>A</i>.</p> <p>If <i>uplo</i> = 'U', only the upper triangular part of <i>A</i> is used to define the elements of the Hermitian matrix.</p> <p>If <i>uplo</i> = 'L', only the lower triangular part of <i>A</i> is used to define the elements of the Hermitian matrix.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, <i>p?heevx</i> cannot guarantee correct error reporting.
<i>v1, vu</i>	<p>(global)</p> <p>REAL for <i>pcheevx</i></p> <p>DOUBLE PRECISION for <i>pzheevx</i>.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il, iu</i>	<p>(global)</p> <p>INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned.</p> <p>Constraints:</p> <p>$il \geq 1; \min(il, n) \leq iu \leq n$.</p> <p>Not referenced if <i>range</i> = 'A' or 'V'.</p>
<i>abstol</i>	<p>(global).</p> <p>REAL for <i>pcheevx</i></p> <p>DOUBLE PRECISION for <i>pzheevx</i>.</p>

If $jobz='V'$, setting $abstol$ to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol+eps*\max(|a|, |b|)$, where eps is the machine precision. If $abstol$ is less than or equal to zero, then $eps*norm(T)$ will be used in its place, where $norm(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2*p?lamch('S')$, not zero. If this routine returns with $((mod(info,2)\neq 0).or.(mod(info/8,2)\neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2*p?lamch('S')$.

NOTE

$mod(x, y)$ is the integer remainder of x/y .

orfac

(global). REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol=orfac*norm(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), tol may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative.

orfac should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

descz

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Z . $descz(ctxt_)$ must equal $desca(ctxt_)$.

work

(local).

COMPLEX for pcheevx

DOUBLE COMPLEX for pzheevx.

Array of size *lwork*.

lwork

(local) INTEGER. The size of the array *work*.

If only eigenvalues are requested:

$$lwork \geq n + \max(nb*(np0 + 1), 3)$$

If eigenvectors are requested:

$$lwork \geq n + (np0+mq0+nb)*nb$$

with $nq0 = \text{numroc}(nn, nb, 0, 0, \text{NPCOL})$.

$$lwork \geq 5*n + \max(5*nn, np0*mq0+2*nb*nb) + \text{iceil}(neig, \text{NPROW}*NPCOL) * nn$$

For optimal performance, greater workspace is needed, that is

$$lwork \geq \max(lwork, nhetrd_lwork)$$

where *lwork* is as defined above, and $nhetrd_lwork = n + 2*(anb + 1)*(4*nps+2) + (nps+1)*nps$

$$ictxt = \text{desca}(ctxt_)$$

$$anb = \text{pjlaenv}(ictxt, 3, 'pchettrd', 'L', 0, 0, 0, 0)$$

$$sqnpc = \text{sqr}t(\text{dble}(\text{NPROW} * \text{NPCOL}))$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2*anb)$$

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pxerbla*.

rwork

(local)

REAL for *pcheevx*

DOUBLE PRECISION for *pzheevx*.

Workspace array of size *lwork*.

lwork

(local) INTEGER. The size of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*nn+4*n$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lwork \geq 4*n + \max(5*nn, np0*mq0+2*nb*nb) + \text{iceil}(neig, \text{NPROW}*NPCOL) * nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following values to *lwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid w(j+1) \leq w(j) + orfac * 2 * \text{norm}(A)\}.$$

Variable definitions:

neig = number of eigenvectors requested;

$$nb = \text{desca}(mb_) = \text{desca}(nb_) = \text{descz}(mb_) = \text{descz}(nb_);$$

$$nn = \max(n, \text{NB}, 2);$$

$$\text{desca}(rsrc_) = \text{desca}(nb_) = \text{descz}(rsrc_) = \text{descz}(csrc_) = 0;$$

$$np0 = \text{numroc}(nn, nb, 0, 0, \text{NPROW});$$

```
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL);
```

`iceil(x, y)` is a ScaLAPACK function returning $\text{ceiling}(x/y)$

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, `p?heevx` attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', `p?heevx` does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow `p?heevx` to compute the eigenvalues, `p?heevx` will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If $\text{clustersize} \geq n/\text{sqrt}(\text{NPROW} * \text{NPCOL})$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, $\text{clustersize} = n-1$) `p?stein` will perform no better than `?stein` on 1 processor.

For $\text{clustersize} = n/\text{sqrt}(\text{NPROW} * \text{NPCOL})$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $\text{clustersize} > n/\text{sqrt}(\text{NPROW} * \text{NPCOL})$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, size of *iwork*.

$liwork \geq 6 * nnp$

Where: $nnp = \max(n, \text{NPROW} * \text{NPCOL} + 1, 4)$

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m (global) INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.

nz (global) INTEGER. Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of *z* that are filled.

If $jobz \neq 'V'$, nz is not referenced.

If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and $p?heevx$ is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z ($m \leq descz(n_)$) and sufficient workspace to compute them. (See $lwork$). $p?heevx$ is always able to detect insufficient space without computation unless $range = 'V'$.

w

(global).

REAL for $pcheevx$

DOUBLE PRECISION for $pzheevx$.

Array of size n . The first m elements contain the selected eigenvalues in ascending order.

z

(local).

COMPLEX for $pcheevx$

DOUBLE COMPLEX for $pzheevx$.

Array, global size (n, n) , local size $(lld_z, LOCC(jz+n-1))$.

If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

$work(1)$

On exit, returns adequate workspace to allow optimal performance.

$rwork$

(local).

REAL for $pcheevx$

DOUBLE PRECISION for $pzheevx$.

Array of size $lwork$. On return, $rwork(1)$ contains the optimal amount of workspace required for efficient execution.

If $jobz = 'N'$ $rwork(1) =$ optimal amount of workspace required to compute eigenvalues efficiently.

If $jobz = 'V'$ $rwork(1) =$ optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If $range = 'V'$, it is assumed that all eigenvectors may be required.

$iwork(1)$

(local)

On return, $iwork(1)$ contains the amount of integer workspace required.

$ifail$

(global) INTEGER.

Array of size n .

If $jobz = 'V'$, then on normal exit, the first m elements of $ifail$ are zero. If $(\text{mod}(info, 2) \neq 0)$ on exit, then $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

iclustr

(global) INTEGER.

Array of size $2 * NPROW * NPCOL$.

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed $iclustr(2*i-1)$ to $iclustr(2*i)$, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. $iclustr()$ is a zero terminated array. $(iclustr(2*k) \neq 0$ and $iclustr(2*k+1) = 0)$ if and only if k is the number of clusters. $iclustr$ is not referenced if $jobz = 'N'$.

gap

(global)

REAL for *pcheevx*

DOUBLE PRECISION for *pzheevx*.

Array of size $(NPROW * NPCOL)$

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C * n) / gap(i)$ where C is a small constant.

info

(global) INTEGER.

If $info = 0$, the execution is successful.

If $info < 0$:

If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$. If the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$:

If $(\text{mod}(info, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure $abstol = 2.0 * p?lamch('U')$

If $(\text{mod}(info/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(info/4, 2) \neq 0)$, then space limit prevented *p?syevx* from computing all of the eigenvectors between vl and vu . The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(info/8, 2) \neq 0)$, then *p?stebz* failed to compute eigenvalues. Ensure $abstol = 2.0 * p?lamch('U')$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
call psgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, info)
```

```
call pdgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, info)
```

```
call pcgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, rwork, info)
```

```
call pzgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt, descvt, work, lwork, rwork, info)
```

Include Files

Description

The p?gesvd routine computes the singular value decomposition (SVD) of an m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix that is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A and the columns of U and V are the corresponding right and left singular vectors, respectively. The singular values are returned in array s in decreasing order and only the first $\min(m, n)$ columns of U and rows of $vt = V^T$ are computed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

mp = number of local rows in A and U

nq = number of local columns in A and VT

$size$ = $\min(m, n)$

$sizeq$ = number of local columns in U

$sizep$ = number of local rows in VT

$jobu$ (global) CHARACTER*1. Specifies options for computing all or part of the matrix U .

	<p>If $jobu = 'V'$, the first $size$ columns of U (the left singular vectors) are returned in the array u;</p> <p>If $jobu = 'N'$, no columns of U (no left singular vectors) are computed.</p>
<i>jobvt</i>	<p>(global) CHARACTER*1.</p> <p>Specifies options for computing all or part of the matrix V^T.</p> <p>If $jobvt = 'V'$, the first $size$ rows of V^T (the right singular vectors) are returned in the array vt;</p> <p>If $jobvt = 'N'$, no rows of V^T (no right singular vectors) are computed.</p>
<i>m</i>	(global) INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in A ($n \geq 0$).
<i>a</i>	<p>(local). REAL for <code>psgesvd</code></p> <p>DOUBLE PRECISION for <code>pdgesvd</code></p> <p>COMPLEX for <code>pcgesvd</code></p> <p>COMPLEX*16 for <code>pzgesvd</code></p> <p>Block cyclic array, global size (m, n), local size (mp, nq).</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>iu, ju</i>	(global) INTEGER. The row and column indices in the global matrix U indicating the first row and the first column of the submatrix U , respectively.
<i>descu</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix U .
<i>ivt, jvt</i>	(global) INTEGER. The row and column indices in the global matrix V^T indicating the first row and the first column of the submatrix V^T , respectively.
<i>descvt</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V^T .
<i>work</i>	<p>(local). REAL for <code>psgesvd</code></p> <p>DOUBLE PRECISION for <code>pdgesvd</code></p> <p>COMPLEX for <code>pcgesvd</code></p> <p>COMPLEX*16 for <code>pzgesvd</code></p> <p>Workspace array of size $lwork$</p>
<i>lwork</i>	<p>(local) INTEGER. The size of the array $work$;</p> <p>$lwork > 2 + 6 * sizeb + \max(watobd, wbdtosvd)$,</p>

where $sizeb = \max(m, n)$, and *watobd* and *wbdtosvd* refer, respectively, to the workspace required to bidiagonalize the matrix *A* and to go from the bidiagonal matrix to the singular value decomposition *USVT*.

For *watobd*, the following holds:

$$watobd = \max(\max(wp?lange, wp?gebrd), \max(wp?lared2d, wp?lared1d)),$$

where *wp?lange*, *wp?lared1d*, *wp?lared2d*, *wp?gebrd* are the workspaces required respectively for the subprograms *p?lange*, *p?lared1d*, *p?lared2d*, *p?gebrd*. Using the standard notation

$$mp = \text{numroc}(m, mb, \text{MYROW}, \text{desca}(\text{ctxt}_), \text{NPROW}),$$

$$nq = \text{numroc}(n, nb, \text{MYCOL}, \text{desca}(\text{lld}_), \text{NPCOL}),$$

the workspaces required for the above subprograms are

$$wp?lange = mp,$$

$$wp?lared1d = nq0,$$

$$wp?lared2d = mp0,$$

$$wp?gebrd = nb * (mp + nq + 1) + nq,$$

where *nq0* and *mp0* refer, respectively, to the values obtained at *MYCOL* = 0 and *MYROW* = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0):

$$watobd \leq nb * (mp0 + nq0 + 1) + nq0.$$

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.

For *wbdtosvd*, the following holds:

$$wbdtosvd = size * (wantu * nru + wantvt * ncvt) + \max(w?bdsqr, \max(wantu * wp?ormbrqln, wantvt * wp?ormbrprt)),$$

where

wantu(wantvt) = 1, if left/right singular vectors are wanted, and *wantu(wantvt)* = 0, otherwise. *w?bdsqr*, *wp?ormbrqln*, and *wp?ormbrprt* refer respectively to the workspace required for the subprograms *?bdsqr*, *p?ormbr(qln)*, and *p?ormbr(prt)*, where *qln* and *prt* are the values of the arguments *vect*, *side*, and *trans* in the call to *p?ormbr*. *nru* is equal to the local number of rows of the matrix *U* when distributed 1-dimensional "column" of processes. Analogously, *ncvt* is equal to the local number of columns of the matrix *VT* when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure *?bdsqr* requires

$$w?bdsqr = \max(1, 2 * size + (2 * size - 4) * \max(wantu, wantvt))$$

on every processor. Finally,

$$wp?ormbrqln = \max((nb * (nb - 1)) / 2, (sizeq + mp) * nb) + nb * nb,$$

$$wp?ormbrprt = \max((mb * (mb - 1)) / 2, (sizep + nq) * mb) + mb * mb,$$

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by *pxerbla*.

rwork REAL for `pcgesvd`
 DOUBLE PRECISION for `pzgesvd`
 Workspace array of size $1 + 4*sizeb$. Not used for `psgesvd` and `pdgesvd`.

Output Parameters

a On exit, the contents of *a* are destroyed.

s (global). REAL for `psgesvd` and `pcgesvd`
 DOUBLE PRECISION for `pdgesvd` and `pzgesvd`
 Array of size *size*.
 Contains the singular values of *A* sorted so that $s(i) \geq s(i+1)$.

u (local). REAL for `psgesvd`
 DOUBLE PRECISION for `pdgesvd`
 COMPLEX for `pcgesvd`
 COMPLEX*16 for `pzgesvd`
 local size (*mp*, *sizeq*), global size (*m*, *size*)
 If *jobu* = 'V', *u* contains the first $\min(m, n)$ columns of *U*.
 If *jobu* = 'N' or 'O', *u* is not referenced.

vt (local). REAL for `psgesvd`
 DOUBLE PRECISION for `pdgesvd`
 COMPLEX for `pcgesvd`
 COMPLEX*16 for `pzgesvd`
 local size (*sizep*, *nq*), global size (*size*, *n*)
 If *jobvt* = 'V', *vt* contains the first *size* rows of V^T if *jobu* = 'N', *vt* is not referenced.

work On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

rwork On exit, if *info* = 0, then *rwork*(1) returns the required size of *rwork*.

info (global) INTEGER.
 If *info* = 0, the execution is successful.
 If *info* < 0, If *info* = -*i*, the *i*th parameter had an illegal value.
 If *info* > 0 *i*, then if `?bdsqr` did not converge,
 If *info* = $\min(m, n) + 1$, then `p?gesvd` has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from `p?gesvd` cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
call pssygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
ifail, iclustr, gap, info)
```

```
call pdsygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
ifail, iclustr, gap, info)
```

Include Files

Description

The p?sygvx routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x, \text{ sub}(A) \text{ sub}(B) * x = \lambda * x, \text{ or } \text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

Here x denotes eigen vectors, λ (*lambda*) denotes eigenvalues, $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$ is assumed to symmetric, and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

- ibtype* (global) INTEGER. Must be 1 or 2 or 3.
 Specifies the problem type to be solved:
 If *ibtype* = 1, the problem type is $\text{sub}(A) * x = \text{lambda} * \text{sub}(B) * x$;
 If *ibtype* = 2, the problem type is $\text{sub}(A) * \text{sub}(B) * x = \text{lambda} * x$;
 If *ibtype* = 3, the problem type is $\text{sub}(B) * \text{sub}(A) * x = \text{lambda} * x$.
- jobz* (global) CHARACTER*1. Must be 'N' or 'V'.
 If *jobz* = 'N', then compute eigenvalues only.
 If *jobz* = 'V', then compute eigenvalues and eigenvectors.
- range* (global) CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues in the interval: [*vl*, *vu*]

If *range* = 'I', the routine computes eigenvalues with indices *il* through *iu*.

uplo

(global) CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of sub(*A*) and sub(*B*);

If *uplo* = 'L', arrays *a* and *b* store the lower triangles of sub(*A*) and sub(*B*).

n

(global) INTEGER. The order of the matrices sub(*A*) and sub(*B*), $n \geq 0$.

a

(local)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Pointer into the local memory to an array of size (*lld_a*, *LOCc(ja+n-1)*). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ia, ja

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*. If *desca(ctxt_)* is incorrect, p?sygvx cannot guarantee correct error reporting.

b

(local). REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Pointer into the local memory to an array of size (*lld_b*, *LOCc(jb+n-1)*). On entry, this array contains the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*B*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*B*) contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix.

ib, jb

(global) INTEGER. The row and column indices in the global matrix *B* indicating the first row and the first column of the submatrix *B*, respectively.

<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>. <i>descb(ctxt_)</i> must be equal to <i>desca(ctxt_)</i>.</p>
<i>vl, vu</i>	<p>(global)</p> <p>REAL for pssygvx</p> <p>DOUBLE PRECISION for pdsygvx.</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>(global)</p> <p>INTEGER.</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1, \min(il, n) \leq iu \leq n$</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>(global)</p> <p>REAL for pssygvx</p> <p>DOUBLE PRECISION for pdsygvx.</p> <p>If <i>jobz</i>='V', setting <i>abstol</i> to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors.</p> <p>The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to</p> $abstol + eps * \max(a , b),$ <p>where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form.</p> <p>Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $((\text{mod}(info, 2) \neq 0) \text{ or } (\text{mod}(info/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2 * p?lamch('S')$.</p>
<hr/> <p>NOTE</p> <p>$\text{mod}(x, y)$ is the integer remainder of x/y.</p> <hr/>	
<i>orfac</i>	<p>(global).</p> <p>REAL for pssygvx</p> <p>DOUBLE PRECISION for pdsygvx.</p> <p>Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be</p>

reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local)

REAL for *pssygvx*DOUBLE PRECISION for *pdsygvx*.Workspace array of size *lwork**lwork*

(local) INTEGER.

Size of the array *work*. See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*n + \max(5*nn, NB*(np0 + 1))$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*nb*nb) + \text{iceil}(neig, NPROW*NPCOL)*nn.$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid w(j+1) \leq w(j) + orfac*2*norm(A)\}$$

Variable definitions:

neig = number of eigenvectors requested,*nb* = *desca(mb_)* = *desca(nb_)* = *descz(mb_)* = *descz(nb_)*,*nn* = max(*n*, *nb*, 2),*desca(rsrc_)* = *desca(nb_)* = *descz(rsrc_)* = *descz(csrc_)* = 0,*np0* = numroc(*nn*, *nb*, 0, 0, NPROW),*mq0* = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, NPCOL)*iceil*(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

$lwork \geq \max(lwork, 5*n + nsytrd_lwopt, nsygst_lwopt)$, where

lwork, as defined previously, depends upon the number of eigenvectors requested, and

$nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps+3)*nps$

$nsygst_lwopt = 2*np0*nb + nq0*nb + nb*nb$

$anb = pjlavenv(desca(ctxt_), 3, p?sytrd, 'L', 0, 0, 0, 0)$

$sqnpc = \text{int}(\text{sqrt}(\text{dble}(\text{NPROW} * \text{NPCOL})))$

$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2*anb)$

$NB = \text{desca}(mb_)$

$np0 = \text{numroc}(n, nb, 0, 0, \text{NPROW})$

$nq0 = \text{numroc}(n, nb, 0, 0, \text{NPCOL})$

numroc is a ScaLAPACK tool functions;

pjlavenv is a ScaLAPACK environmental inquiry function

MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If $clustersize \geq n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, $clustersize = n-1$) *p?stein* will perform no better than *?stein* on a single processor.

For $clustersize = n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $clustersize > n/\text{sqrt}(\text{NPROW}*\text{NPCOL})$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pxerbla*.

iwork

(local) INTEGER. Workspace array.

liwork (local) INTEGER , size of *iwork*.

$liwork \geq 6 * nnp$

Where:

$nnp = \max(n, NPROW * NPCOL + 1, 4)$

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit,
If *jobz* = 'V', and if *info* = 0, sub(A) contains the distributed matrix Z of eigenvectors. The eigenvectors are normalized as follows:

for *ibtype* = 1 or 2, $Z^T * \text{sub}(B) * Z = I$;

for *ibtype* = 3, $Z^T * \text{inv}(\text{sub}(B)) * Z = I$.

If *jobz* = 'N', then on exit the upper triangle (if *uplo*='U') or the lower triangle (if *uplo*='L') of sub(A), including the diagonal, is destroyed.

b On exit, if *info* ≤ *n*, the part of sub(B) containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $\text{sub}(B) = U^T * U$ or $\text{sub}(B) = L * L^T$.

m (global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

nz (global) INTEGER.

Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of z that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and `p?sygvx` is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* ($m \leq \text{descz}(n)$) and sufficient workspace to compute them. (See *lwork* below.) `p?sygvx` is always able to detect insufficient space without computation unless *range*='V'.

w (global)

REAL for `pssygvx`

DOUBLE PRECISION for `pdsygvx`.

Array of size *n*. On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z (local).

REAL for `pssygvx`

DOUBLE PRECISION for `pdsygvx`.

global size (*n*, *n*), local size (*lld_z*, *LOC*(*jz*+*n*-1)).

If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

work

If $jobz='N'$ $work(1) =$ optimal amount of workspace required to compute eigenvalues efficiently

If $jobz = 'V'$ $work(1) =$ optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If $range='V'$, it is assumed that all eigenvectors may be required.

ifail

(global) INTEGER.

Array of size n .

ifail provides additional information when $info \neq 0$

If $(\text{mod}(info/16,2) \neq 0)$ then $ifail(1)$ indicates the order of the smallest minor which is not positive definite. If $(\text{mod}(info,2) \neq 0)$ on exit, then $ifail$ contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and $jobz = 'V'$, then the first m elements of $ifail$ are set to zero.

iclustr

(global) INTEGER.

Array of size $(2 * NPROW * NPCOL)$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed $iclustr(2*i-1)$ to $iclustr(2*i)$, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. $iclustr()$ is a zero terminated array.

$(iclustr(2*k) \neq 0 \text{ and } iclustr(2*k+1) = 0)$ if and only if k is the number of clusters $iclustr$ is not referenced if $jobz = 'N'$.

gap

(global)

REAL for *pssygvx*

DOUBLE PRECISION for *pdsygvx*.

Array of size $NPROW * NPCOL$. This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C * n) / gap(i)$, where C is a small constant.

info

(global) INTEGER.

If $info = 0$, the execution is successful.

If $info < 0$: the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$:

If $(\text{mod}(info, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(info, 2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(info/4, 2) \neq 0)$, then space limit prevented $p?sygvx$ from computing all of the eigenvectors between vl and vu . The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(info/8, 2) \neq 0)$, then $p?stebz$ failed to compute eigenvalues.

If $(\text{mod}(info/16, 2) \neq 0)$, then B was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

$p?hegvx$

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
call pchegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
iwork, liwork, ifail, iclustr, gap, info)
```

```
call pzhegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
iwork, liwork, ifail, iclustr, gap, info)
```

Include Files

Description

The $p?hegvx$ routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x, \quad \text{sub}(A) * \text{sub}(B) * x = \lambda * x, \quad \text{or} \quad \text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

Here $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ are assumed to be Hermitian and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

Input Parameters

<i>ibtype</i>	<p>(global) INTEGER. Must be 1 or 2 or 3.</p> <p>Specifies the problem type to be solved:</p> <p>If <i>ibtype</i> = 1, the problem type is $\text{sub}(A) * x = \text{lambda} * \text{sub}(B) * x;$</p> <p>If <i>ibtype</i> = 2, the problem type is $\text{sub}(A) * \text{sub}(B) * x = \text{lambda} * x;$</p> <p>If <i>ibtype</i> = 3, the problem type is $\text{sub}(B) * \text{sub}(A) * x = \text{lambda} * x.$</p>
<i>jobz</i>	<p>(global) CHARACTER*1. Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>(global) CHARACTER*1. Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues in the interval: [<i>vl</i>, <i>vu</i>]</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> through <i>iu</i>.</p>
<i>uplo</i>	<p>(global) CHARACTER*1. Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>sub(A)</i> and <i>sub(B)</i>;</p> <p>If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>sub(A)</i> and <i>sub(B)</i>.</p>
<i>n</i>	<p>(global) INTEGER.</p> <p>The order of the matrices <i>sub(A)</i> and <i>sub(B)</i> ($n \geq 0$).</p>
<i>a</i>	<p>(local)</p> <p>COMPLEX for <i>pchegvx</i></p> <p>DOUBLE COMPLEX for <i>pzhgvx</i>.</p> <p>Pointer into the local memory to an array of size (<i>lld_a</i>, <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> Hermitian distributed matrix <i>sub(A)</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>sub(A)</i> contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>sub(A)</i> contains the lower triangular part of the matrix.</p>
<i>ia, ja</i>	<p>(global) INTEGER.</p>

The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix A . If *desca(ctxt_)* is incorrect, *p?hegvx* cannot guarantee correct error reporting.

b (local).

COMPLEX for *pchegvx*

DOUBLE COMPLEX for *pzhgvx*.

Pointer into the local memory to an array of size $(lld_b, LOC(jb+n-1))$. On entry, this array contains the local pieces of the n -by- n Hermitian distributed matrix $sub(B)$.

If *uplo* = 'U', the leading n -by- n upper triangular part of $sub(B)$ contains the upper triangular part of the matrix.

If *uplo* = 'L', the leading n -by- n lower triangular part of $sub(B)$ contains the lower triangular part of the matrix.

ib, jb (global) INTEGER.

The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.

descb (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix B . *descb(ctxt_)* must be equal to *desca(ctxt_)*.

vl, vu (global)

REAL for *pchegvx*

DOUBLE PRECISION for *pzhgvx*.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu (global)

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1, \min(il, n) \leq iu \leq n$

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol (global)

REAL for *pchegvx*

DOUBLE PRECISION for *pzhgvx*.

If *jobz*='V', setting *abstol* to *p?lamch(context, 'U')* yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to

$$abstol + eps * \max(|a|, |b|),$$

where eps is the machine precision. If $abstol$ is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $((\text{mod}(\text{info}, 2) \neq 0) .or. * (\text{mod}(\text{info}/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2 * p?lamch('S')$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

orfac

(global).

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), tol may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0E-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

descz

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Z . $descz(ctxt_)$ must equal $desca(ctxt_)$.

work

(local)

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

Workspace array of size *lwork*

lwork

(local).

INTEGER. The size of the array *work*.

If only eigenvalues are requested:

$$lwork \geq n + \max(NB * (np0 + 1), 3)$$

If eigenvectors are requested:

$$lwork \geq n + (np0 + mq0 + NB) * NB$$

with $nq0 = \text{numroc}(nn, NB, 0, 0, NPCOL)$.

For optimal performance, greater workspace is needed, that is

$$lwork \geq \max(lwork, n, nhetr_lwork, nhegst_lwork)$$

where *lwork* is as defined above, and

$$nhetr_lwork = 2*(anb+1)*(4*nps+2) + (nps + 1)*nps;$$

$$nhegst_lwork = 2*np0*nb + nq0*nb + nb*nb$$

$$nb = desca(mb_)$$

$$np0 = \text{numroc}(n, nb, 0, 0, \text{NPROW})$$

$$nq0 = \text{numroc}(n, nb, 0, 0, \text{NPCOL})$$

$$ictxt = desca(ctxt_)$$

$$anb = \text{pjlaenv}(ictxt, 3, 'p?hettrd', 'L', 0, 0, 0, 0)$$

$$sqnpc = \text{sqrt}(\text{dble}(\text{NPROW} * \text{NPCOL}))$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2*anb)$$

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pxerbla*.

rwork

(local)

REAL for *pchegvx*

DOUBLE PRECISION for *pzhgevx*.

Workspace array of size *lrwork*.

lrwork

(local) INTEGER. The size of the array *rwork*.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lrwork \geq 5*nn+4*n$

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lrwork \geq 4*n + \max(5*nn, np0*mq0) + \text{iceil}(neig, \text{NPROW} * \text{NPCOL}) * nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lrwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w(k), \dots, w(k+clustersize-1) \mid |w(j+1) - w(j)| \leq orfac * 2 * \text{norm}(A)\}$$

Variable definitions:

```

neig = number of eigenvectors requested;
nb = desca(mb_) = desca(nb_) = descz(mb_) = descz(nb_);
nn = max(n, nb, 2);
desca(rsrc_) = desca(nb_) = descz(rsrc_) = descz(csrc_) = 0;
np0 = numroc(nn, nb, 0, 0, NPROW);
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL);
iceil(x, y) is a ScaLAPACK function returning ceiling(x/y).

```

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, p?hegvx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range*='V', p?hegvx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?hegvx to compute the eigenvalues, p?hegvx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If *clustersize* > $n/\sqrt{NPROW*NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = $n-1$) p?stein will perform no better than ?stein on 1 processor.

For *clustersize* = $n/\sqrt{NPROW*NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* > $n/\sqrt{NPROW*NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by p?erbla.

iwork (local) INTEGER. Workspace array.

liwork (local) INTEGER, size of *iwork*.

$liwork \geq 6 * nnp$

Where: $nnp = \max(n, NPROW*NPCOL + 1, 4)$

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by p?erbla.

Output Parameters

<i>a</i>	<p>On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, sub(A) contains the distributed matrix <i>Z</i> of eigenvectors.</p> <p>The eigenvectors are normalized as follows:</p> <p>If <i>ibtype</i> = 1 or 2, then $Z^H \text{sub}(B) * Z = I$;</p> <p>If <i>ibtype</i> = 3, then $Z^H \text{inv}(\text{sub}(B)) * Z = I$.</p> <p>If <i>jobz</i> = 'N', then on exit the upper triangle (if <i>uplo</i>='U') or the lower triangle (if <i>uplo</i>='L') of sub(A), including the diagonal, is destroyed.</p>
<i>b</i>	<p>On exit, if <i>info</i> ≤ <i>n</i>, the part of sub(B) containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $\text{sub}(B) = U^H * U$, or $\text{sub}(B) = L * L^H$.</p>
<i>m</i>	(global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.
<i>nz</i>	<p>(global) INTEGER. Total number of eigenvectors computed. $0 < nz < m$. The number of columns of <i>z</i> that are filled.</p> <p>If <i>jobz</i> ≠ 'V', <i>nz</i> is not referenced.</p> <p>If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i> unless the user supplies insufficient space and p?hegvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in <i>z</i> ($m \leq \text{descz}(n_)$) and sufficient workspace to compute them. (See <i>lwork</i> below.) The routine p?hegvx is always able to detect insufficient space without computation unless <i>range</i> = 'V'.</p>
<i>w</i>	<p>(global)</p> <p>REAL for pchegvx</p> <p>DOUBLE PRECISION for pzhegvx.</p> <p>Array of size <i>n</i>. On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.</p>
<i>z</i>	<p>(local).</p> <p>COMPLEX for pchegvx</p> <p>DOUBLE COMPLEX for pzhegvx.</p> <p>global size (<i>n</i>, <i>n</i>), local size (<i>lld_z</i>, <i>LOCc(jz+n-1)</i>).</p> <p>If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work</i>	On exit, <i>work</i> (1) returns the optimal amount of workspace.
<i>rwork</i>	On exit, <i>rwork</i> (1) contains the amount of workspace required for optimal efficiency

If *jobz*='N' *rwork*(1) = optimal amount of workspace required to compute eigenvalues efficiently

If *jobz*='V' *rwork*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required when computing optimal workspace.

ifail

(global) INTEGER.

Array of size *n*.

ifail provides additional information when *info*≠0

If $(\text{mod}(\text{info}/16,2) \neq 0)$, then *ifail*(1) indicates the order of the smallest minor which is not positive definite.

If $(\text{mod}(\text{info},2) \neq 0)$ on exit, then *ifail*(1) contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions are held, and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr

(global) INTEGER.

Array of size $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2**i*-1) to *iclustr*(2**i*), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

iclustr() is a zero terminated array. (*iclustr*(2**k*) ≠ 0 .and. *iclustr*(2**k*+1)=0) if and only if *k* is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap

(global)

REAL for *pchegvx*

DOUBLE PRECISION for *pzhgevx*.

Array of size *NPROW***NPCOL*.

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C * n) / \text{gap}(i)$, where *C* is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i * 100 + j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

If $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(\text{info}, 2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented `p?sygvx` from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then `p?stebz` failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2) \neq 0)$, then *B* was not positive definite. *ifail(1)* indicates the order of the smallest minor which is not positive definite.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Auxiliary, Utility, and Redistribution/Copy Routines

5

This chapter describes the Intel® Math Kernel Library implementation of ScaLAPACK [Auxiliary Routines](#), [Utility Functions and Routines](#), and [Matrix Redistribution/Copy Routines](#). The library includes routines for both real and complex data.

NOTE

ScaLAPACK routines are provided only for Intel® 64 or Intel® Many Integrated Core architectures.

Routine naming conventions, mathematical notation, and matrix storage schemes used for ScaLAPACK auxiliary and utility routines are the same as described in previous chapters. Some routines and functions may have combined character codes, such as `sc` or `dz`. For example, the routine `pzcsum1` uses a complex input array and returns a real value.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Auxiliary Routines

ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
<code>b?laapp</code>	<code>s, d</code>	Multiplies a matrix with an orthogonal matrix.
<code>b?laexc</code>	<code>s, d</code>	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
<code>b?trexc</code>	<code>s, d</code>	Reorders the Schur factorization of a general matrix.
<code>p?lacgv</code>	<code>c, z</code>	Conjugates a complex vector.
<code>p?max1</code>	<code>c, z</code>	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS <code>p?amax</code> , but using the absolute value to the real part).
<code>pmpcol</code>	<code>s, d</code>	<i>Finds the collaborators of a process.</i>
<code>pmpim2</code>	<code>s, d</code>	Computes the eigenpair range assignments for all processes.

Routine Name	Data Types	Description
<code>?combamax1</code>	<code>c, z</code>	Finds the element with maximum real part absolute value and its corresponding global index.
<code>p?sum1</code>	<code>sc, dz</code>	Forms the 1-norm of a complex vector similar to Level 1 PBLAS <code>p?asum</code> , but using the true absolute value.
<code>p?dbtrsv</code>	<code>s, d, c, z</code>	Computes an <i>LU</i> factorization of a general tridiagonal matrix with no pivoting. The routine is called by <code>p?dbtrs</code> .
<code>p?dttrsv</code>	<code>s, d, c, z</code>	Computes an <i>LU</i> factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by <code>p?dttrs</code> .
<code>p?gebal</code>	<code>s, d</code>	Balances a general real matrix.
<code>p?gebd2</code>	<code>s, d, c, z</code>	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
<code>p?gehd2</code>	<code>s, d, c, z</code>	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
<code>p?gelq2</code>	<code>s, d, c, z</code>	Computes an <i>LQ</i> factorization of a general rectangular matrix (unblocked algorithm).
<code>p?geql2</code>	<code>s, d, c, z</code>	Computes a <i>QL</i> factorization of a general rectangular matrix (unblocked algorithm).
<code>p?geqr2</code>	<code>s, d, c, z</code>	Computes a <i>QR</i> factorization of a general rectangular matrix (unblocked algorithm).
<code>p?gerq2</code>	<code>s, d, c, z</code>	Computes an <i>RQ</i> factorization of a general rectangular matrix (unblocked algorithm).
<code>p?getf2</code>	<code>s, d, c, z</code>	Computes an <i>LU</i> factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
<code>p?labrd</code>	<code>s, d, c, z</code>	Reduces the first <i>nb</i> rows and columns of a general rectangular matrix <i>A</i> to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of <i>A</i> .
<code>p?lacon</code>	<code>s, d, c, z</code>	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
<code>p?laconsb</code>	<code>s, d</code>	Looks for two consecutive small subdiagonal elements.
<code>p?laccp2</code>	<code>s, d, c, z</code>	Copies all or part of a distributed matrix to another distributed matrix.
<code>p?laccp3</code>	<code>s, d</code>	Copies from a global parallel array into a local replicated array or vice versa.
<code>p?laccpy</code>	<code>s, d, c, z</code>	Copies all or part of one two-dimensional array to another.
<code>p?laevswp</code>	<code>s, d, c, z</code>	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
<code>p?lahrd</code>	<code>s, d, c, z</code>	Reduces the first <i>nb</i> columns of a general rectangular matrix <i>A</i> so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of <i>A</i> .

Routine Name	Data Types	Description
<code>p?laiect</code>	<code>s, d, c, z</code>	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
<code>p?lamve</code>	<code>s, d</code>	Copies all or part of one two-dimensional distributed array to another.
<code>p?lange</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
<code>p?lanhs</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
<code>p?lansy, p?lanhe</code>	<code>s, d, c, z / c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
<code>p?lantr</code>	<code>s, d, c, z</code>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
<code>p?lapiv</code>	<code>s, d, c, z</code>	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
<code>p?laqge</code>	<code>s, d, c, z</code>	Scales a general rectangular matrix, using row and column scaling factors computed by <code>p?geequ</code> .
<code>p?laqr0</code>	<code>s, d</code>	Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.
<code>p?laqr1</code>	<code>s, d</code>	Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.
<code>p?laqr2</code>	<code>s, d</code>	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
<code>p?laqr3</code>	<code>s, d</code>	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
<code>p?laqr4</code>	<code>s, d</code>	Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.
<code>p?laqr5</code>	<code>s, d</code>	Performs a single small-bulge multi-shift QR sweep.
<code>p?laqsy</code>	<code>s, d, c, z</code>	Scales a symmetric/Hermitian matrix, using scaling factors computed by <code>p?poequ</code> .
<code>p?lared1d</code>	<code>s, d</code>	Redistributes an array assuming that the input array <i>bycol</i> is distributed across rows and that all process columns contain the same copy of <i>bycol</i> .
<code>p?lared2d</code>	<code>s, d</code>	Redistributes an array assuming that the input array <i>byrow</i> is distributed across columns and that all process rows contain the same copy of <i>byrow</i> .
<code>p?larf</code>	<code>s, d, c, z</code>	Applies an elementary reflector to a general rectangular matrix.

Routine Name	Data Types	Description
<code>p?larfb</code>	<code>s, d, c, z</code>	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
<code>p?larfc</code>	<code>c, z</code>	Applies the conjugate transpose of an elementary reflector to a general matrix.
<code>p?larfg</code>	<code>s, d, c, z</code>	Generates an elementary reflector (Householder matrix).
<code>p?larft</code>	<code>s, d, c, z</code>	Forms the triangular vector T of a block reflector $H=I- VTV^H$
<code>p?larz</code>	<code>s, d, c, z</code>	Applies an elementary reflector as returned by <code>p?tzzrf</code> to a general matrix.
<code>p?larzb</code>	<code>s, d, c, z</code>	Applies a block reflector or its transpose/conjugate-transpose as returned by <code>p?tzzrf</code> to a general matrix.
<code>p?larzc</code>	<code>c, z</code>	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by <code>p?tzzrf</code> to a general matrix.
<code>p?larzt</code>	<code>s, d, c, z</code>	Forms the triangular factor T of a block reflector $H=I- VTV^H$ as returned by <code>p?tzzrf</code> .
<code>p?lascl</code>	<code>s, d, c, z</code>	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .
<code>p?laset</code>	<code>s, d, c, z</code>	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
<code>p?lasmsub</code>	<code>s, d</code>	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
<code>p?lassq</code>	<code>s, d, c, z</code>	Updates a sum of squares represented in scaled form.
<code>p?laswp</code>	<code>s, d, c, z</code>	Performs a series of row interchanges on a general rectangular matrix.
<code>p?latra</code>	<code>s, d, c, z</code>	Computes the trace of a general square distributed matrix.
<code>p?latrd</code>	<code>s, d, c, z</code>	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
<code>p?latrz</code>	<code>s, d, c, z</code>	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
<code>p?lauu2</code>	<code>s, d, c, z</code>	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices (local unblocked algorithm).
<code>p?lauum</code>	<code>s, d, c, z</code>	Computes the product UU^H or $L^H L$, where U and L are upper or lower triangular matrices.
<code>p?lawil</code>	<code>s, d</code>	Forms the Wilkinson transform.
<code>p?org2l/p?ung2l</code>	<code>s, d, c, z</code>	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by <code>p?geqlf</code> (unblocked algorithm).
<code>p?org2r/p?ung2r</code>	<code>s, d, c, z</code>	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by <code>p?geqrf</code> (unblocked algorithm).
<code>p?orgl2/p?ungl2</code>	<code>s, d, c, z</code>	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by <code>p?gelqf</code> (unblocked algorithm).

Routine Name	Data Types	Description
<code>p?orgr2/p?ungr2</code>	<code>s, d, c, z</code>	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by <code>p?gerqf</code> (unblocked algorithm).
<code>p?orm2l/p?unm2l</code>	<code>s, d, c, z</code>	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by <code>p?geqlf</code> (unblocked algorithm).
<code>p?orm2r/p?unm2r</code>	<code>s, d, c, z</code>	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by <code>p?geqrf</code> (unblocked algorithm).
<code>p?orml2/p?unml2</code>	<code>s, d, c, z</code>	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by <code>p?gelqf</code> (unblocked algorithm).
<code>p?ormr2/p?unmr2</code>	<code>s, d, c, z</code>	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by <code>p?gerqf</code> (unblocked algorithm).
<code>p?pbtrsv</code>	<code>s, d, c, z</code>	Solves a single triangular linear system via <code>frontsolve</code> or <code>backsolve</code> where the triangular matrix is a factor of a banded matrix computed by <code>p?pbtrf</code> .
<code>p?pttrsv</code>	<code>s, d, c, z</code>	Solves a single triangular linear system via <code>frontsolve</code> or <code>backsolve</code> where the triangular matrix is a factor of a tridiagonal matrix computed by <code>p?pttrf</code> .
<code>p?potf2</code>	<code>s, d, c, z</code>	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
<code>p?rot</code>	<code>s, d</code>	Applies a planar rotation to two distributed vectors.
<code>p?rscl</code>	<code>s, d, cs, zd</code>	Multiplies a vector by the reciprocal of a real scalar.
<code>p?sygs2/p?hegs2</code>	<code>s, d, c, z</code>	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from <code>p?potrf</code> (local unblocked algorithm).
<code>p?sytd2/p?hetd2</code>	<code>s, d, c, z</code>	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
<code>p?trord</code>	<code>s, d</code>	Reorders the Schur factorization of a general matrix.
<code>p?trsen</code>	<code>s, d</code>	Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.
<code>p?trti2</code>	<code>s, d, c, z</code>	Computes the inverse of a triangular matrix (local unblocked algorithm).
<code>?lamsh</code>	<code>s, d</code>	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
<code>?laqr6</code>	<code>s, d</code>	Performs a single small-bulge multi-shift QR sweep collecting the transformations.
<code>?lar1va</code>	<code>s, d</code>	<i>Computes scaled eigenvector corresponding to given eigenvalue.</i>
<code>?laref</code>	<code>s, d</code>	Applies Householder reflectors to matrices on either their rows or columns.
<code>?larrb2</code>	<code>s, d</code>	Provides limited bisection to locate eigenvalues for more accuracy.
<code>?larrd2</code>	<code>s, d</code>	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Routine Name	Data Types	Description
?larre2	s, d	Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.
?larre2a	s, d	<i>Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.</i>
?larrf2	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrv2	s, d	Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L , D and the eigenvalues of $L*D*L^T$.
?lasorte	s, d	Sorts eigenpairs by real and complex data types.
?lasrt2	s, d	Sorts numbers in increasing or decreasing order.
?stegr2	s, d	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stegr2a	s, d	Computes selected eigenvalues and initial representations needed for eigenvector computations.
?stegr2b	s, d	From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.
?stein2	s, d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtf2	s, d, c, z	Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s, d, c, z	Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s, d, c, z	Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s, d, c, z	Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf .
?pttrsv	s, d, c, z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by ?pttrf .
?steqr2	s, d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

b?laapp

Multiplies a matrix with an orthogonal matrix.

Syntax

```
call bslaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
```

```
call bdlaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
```

Description

b?laapp computes

$$B = Q^T A \text{ or } B = A Q$$

where A is an m -by- n matrix and Q is an orthogonal matrix represented by the parameters in the arrays *itraf* and *dtraf* as described in [b?trexc](#).

This is an auxiliary routine called by [p?trord](#).

Input Parameters

<i>iside</i>	INTEGER	Specifies whether Q multiplies A from the left or right as follows: = 0: compute $B = Q^T A$; = 1: compute $B = A Q$.
<i>m</i>	INTEGER	The number of rows of A .
<i>n</i>	INTEGER	The number of columns of A .
<i>nb</i>	INTEGER	If <i>iside</i> = 0, the Q is applied block column-wise to the rows of A and <i>nb</i> specifies the maximal width of the block columns. If <i>iside</i> = 1, this variable is not referenced.
<i>a</i>	REAL for bslaapp DOUBLE PRECISION for bdlaapp	Array of size (lda, n) . On entry, the matrix A .
<i>lda</i>	INTEGER	The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.
<i>nitraf</i>	INTEGER	Length of the array <i>itraf</i> . $nitraf \geq 0$.
<i>itraf</i>	INTEGER array, length <i>nitraf</i>	

List of parameters for representing the transformation matrix Q , see [b?trexc](#).

work (workspace) REAL array of size n .

OUTPUT Parameters

a a is overwritten by B .

dtraf REAL for `bslaapp`
DOUBLE PRECISION for `bdlaapp`

Array, length k .

If $iside=0$, $k = 3*(n/nb)$.

If $iside=1$, $k = 3*nitraf$.

List of parameters for representing the transformation matrix Q , see [b?trexc](#)

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

b?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call bslaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
```

```
call bdlaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
```

Description

`b?laexc` swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

In contrast to the LAPACK routine `?laexc`, the orthogonal transformation matrix Q is not explicitly constructed but represented by parameters contained in the arrays *itraf* and *dtraf*. See the description of [b?trexc](#) for more details.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

n INTEGER
The order of the matrix T . $n \geq 0$.

t REAL for `bslaexc`
DOUBLE PRECISION for `bdlaexc`
Array of size (ldt, n) .
The upper quasi-triangular matrix T , in Schur canonical form.

<i>ldt</i>	INTEGER The leading dimension of the array <i>t</i> . $ldt \geq \max(1, n)$.
<i>j1</i>	INTEGER The index of the first row of the first block <i>T11</i> .
<i>n1</i>	INTEGER The order of the first block <i>T11</i> . $n1 = 0, 1$ or 2 .
<i>n2</i>	INTEGER The order of the second block <i>T22</i> . $n2 = 0, 1$ or 2 .
<i>work</i>	REAL for <code>bslaexc</code> DOUBLE PRECISION for <code>bdlaexc</code> (Workspace) array of size <i>n</i> .

OUTPUT Parameters

<i>t</i>	The updated matrix <i>T</i> , in Schur canonical form.
<i>itraf</i>	INTEGER array, length <i>k</i> , where $k = 1$, if $n1+n2 = 2$; $k = 2$, if $n1+n2 = 3$; $k = 4$, if $n1+n2 = 4$. List of parameters for representing the transformation matrix <i>Q</i> , see b?trexc .
<i>dtraf</i>	REAL for <code>bslaexc</code> DOUBLE PRECISION for <code>bdlaexc</code> Array, length <i>k</i> , where $k = 2$, if $n1+n2 = 2$; $k = 5$, if $n1+n2 = 3$; $k = 10$, if $n1+n2 = 4$. List of parameters for representing the transformation matrix <i>Q</i> , see b?trexc .
<i>info</i>	INTEGER = 0: successful exit = 1: the transformed matrix <i>T</i> would be too far from Schur form; the blocks are not swapped and <i>T</i> and <i>Q</i> are unchanged.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

b?trexc

Reorders the Schur factorization of a general matrix.

Syntax

```
call bstrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
```

```
call bdtrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
```

Description

b?trexc reorders the real Schur factorization of a real matrix $A = Q * T * Q^T$, so that the diagonal block of T with row index *ifst* is moved to row *ilst*.

The real Schur form T is reordered by an orthogonal similarity transformation $Z^T * T * Z$. In contrast to the LAPACK routine ?trexc, the orthogonal matrix Z is not explicitly constructed but represented by parameters contained in the arrays *itraf* and *dtraf*. See Application Notes for further details.

T must be in Schur canonical form (as returned by ?hseqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

<i>n</i>	INTEGER	The order of the matrix T . $n \geq 0$.
<i>t</i>	REAL for bstrexc DOUBLE PRECISION for bdtrexc	Array of size (ldt, n) . The upper quasi-triangular matrix T , in Schur canonical form.
<i>ldt</i>	INTEGER	The leading dimension of the array t . $ldt \geq \max(1, n)$.
<i>ifst, ilst</i>	INTEGER	Specify the reordering of the diagonal blocks of T . The block with row index <i>ifst</i> is moved to row <i>ilst</i> , by a sequence of transpositions between adjacent blocks.
<i>nitraf</i>	INTEGER	Length of the array <i>itraf</i> . As a minimum requirement, $nitraf \geq \max(1, ilst - ifst)$. If there are 2-by-2 blocks in t then <i>nitraf</i> must be larger; a safe choice is $nitraf \geq \max(1, 2 * ilst - ifst)$.
<i>ndtraf</i>	INTEGER	Length of the array <i>dtraf</i> . As a minimum requirement, $ndtraf \geq \max(1, 2 * ilst - ifst)$.

If there are 2-by-2 blocks in t then $ndtraf$ must be larger; a safe choice is $ndtraf \geq \max(1, 5 * |ilst - ifst|)$.

work REAL for `bstrexc`
DOUBLE PRECISION for `bdtrexc`
(Workspace) array of size n .

OUTPUT Parameters

t On exit, the reordered upper quasi-triangular matrix, in Schur canonical form.

$ifst, ilst$ If $ifst$ pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by +1 or -1).
 $1 \leq ifst \leq n; 1 \leq ilst \leq n$.

$nitraf$ Actual length of the array $itraf$.

$itraf$ INTEGER array, length $nitraf$
List of parameters for representing the transformation matrix Z . See Application Notes for further details.

$ndtraf$ Actual length of the array $dtraf$.

$dtraf$ REAL for `bstrexc`
DOUBLE PRECISION for `bdtrexc`
Array, length $ndtraf$
List of parameters for representing the transformation matrix Z . See Application Notes for further details.

$info$ INTEGER
= 0: successful exit
< 0: if $info = -i$, the i -th argument had an illegal value
= 1: two adjacent blocks were too close to swap (the problem is very ill-conditioned); t may have been partially reordered, and $ilst$ points to the first row of the current position of the block being moved.
= 2: the 2 by 2 block to be reordered split into two 1 by 1 blocks and the second block failed to swap with an adjacent block. $ilst$ points to the first row of the current position of the whole block being moved.

Application Notes

The orthogonal transformation matrix Z is a product of $nitraf$ elementary orthogonal transformations. The parameters defining these transformations are stored in the arrays $itraf$ and $dtraf$ as follows:

Consider the i -th transformation acting on rows/columns $pos, pos+1, \dots$. If this transformation is

- a Givens rotation with cosine c and sine s then
 $itraf(i) = pos, dtraf(i) = c, dtraf(i+1) = s;$
- a Householder reflector $H = I - t * v * v'$ with $v = [1; v2; v3]$ then

$itraf(i) = n + pos, dtraf(i) = t, dtraf(i+1) = v2, dtraf(i+2) = v3;$

- a Householder reflector $H = I - t * v * v'$ with $v = [v1; v2; 1]$ then

$itraf(i) = 2*n + pos, dtraf(i) = v1, dtraf(i+1) = v2, dtraf(i+2) = t;$

Note that the parameters in *dtraf* are stored consecutively.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacgv

Conjugates a complex vector.

Syntax

```
call pclacgv(n, x, ix, jx, descx, incx)
```

```
call pzlacgv(n, x, ix, jx, descx, incx)
```

Description

The p?lacgv routine conjugates a complex vector *sub(X)* of length *n*, where *sub(X)* denotes $X(ix, jx:jx+n-1)$ if *incx* = *m_x*, and $X(ix:ix+n-1, jx)$ if *incx* = 1.

Input Parameters

<i>n</i>	(global) INTEGER. The length of the distributed vector <i>sub(X)</i> .
<i>x</i>	(local). COMPLEX for pclacgv COMPLEX*16 for pzlacgv. Pointer into the local memory to an array of size (<i>lld_x</i> , *). On entry the vector to be conjugated $x(i) = X(ix+(jx-1)*m_x+(i-1)*incx), 1 \leq i \leq n$.
<i>ix</i>	(global) INTEGER. The row index in the global matrix <i>X</i> indicating the first row of <i>sub(X)</i> .
<i>jx</i>	(global) INTEGER. The column index in the global matrix <i>X</i> indicating the first column of <i>sub(X)</i> .
<i>descx</i>	(global and local) INTEGER. Array of size <i>dlen_</i> =9. The array descriptor for the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. The global increment for the elements of <i>X</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	(local). On exit, the local pieces of conjugated distributed vector <i>sub(X)</i> .
----------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax, but using the absolute value to the real part).

Syntax

```
call p?max1(n, amax, indx, x, ix, jx, descx, incx)
```

```
call pzmax1(n, amax, indx, x, ix, jx, descx, incx)
```

Description

The p?max1 routine computes the global index of the maximum element in absolute value of a distributed vector sub(X). The global index is returned in *indx* and the value is returned in *amax*, where sub(X) denotes $X(ix:ix+n-1, jx)$ if *incx* = 1, $X(ix, jx:jx+n-1)$ if *incx* = m_x .

Input Parameters

<i>n</i>	(global) pointer to INTEGER. The number of components of the distributed vector sub(X). $n \geq 0$.
<i>x</i>	(local) COMPLEX for p?max1. COMPLEX*16 for pzmax1 Pointer into the local memory to an array of size (<i>lld_x</i> , $LOCc(jx+n-1)$). On entry this array contains the local pieces of the distributed vector sub(X).
<i>ix</i>	(global) INTEGER. The row index in the global matrix X indicating the first row of sub(X).
<i>jx</i>	(global) INTEGER. The column index in the global matrix X indicating the first column of sub(X).
<i>descx</i>	(global and local) INTEGER. Array of size <i>dlen_</i> . The array descriptor for the distributed matrix X .
<i>incx</i>	(global) INTEGER. The global increment for the elements of X . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>amax</i>	(global output) pointer to REAL. The absolute value of the largest entry of the distributed vector sub(X) only in the scope of sub(X).
<i>indx</i>	(global output) pointer to INTEGER. The global index of the element of the distributed vector sub(X) whose real part has maximum absolute value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

pmpcol

Finds the collaborators of a process.

Syntax

```
call pmpcol( myproc, nprocs, iil, needil, neediu, pmyils, pmyius, colbrt, frstcl, lastcl )
```

Description

Using the output from [pmpim2](#) and given the information on eigenvalue clusters, `pmpcol` finds the collaborators of `myproc`.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<code>myproc</code>	INTEGER	The processor number, $0 \leq myproc < nprocs$.
<code>nprocs</code>	INTEGER	The total number of processors available.
<code>iil</code>	INTEGER	The index of the leftmost eigenvalue in the eigenvalue cluster.
<code>needil</code>	INTEGER	The leftmost position in the eigenvalue cluster needed by <code>myproc</code> .
<code>neediu</code>	INTEGER	The rightmost position in the eigenvalue cluster needed by <code>myproc</code> .
<code>pmyils</code>	INTEGER array	For each processor p , $0 < p \leq nprocs$, <code>pmyils(p)</code> is the index of the first eigenvalue in the eigenvalue cluster to be computed. <code>pmyils(p)</code> equals zero if p stays idle.
<code>pmyius</code>	INTEGER array	For each processor p , <code>pmyius(p)</code> is the index of the last eigenvalue in the eigenvalue cluster to be computed.

$pmyius(p)$ equals zero if p stays idle.

OUTPUT Parameters

colbrt LOGICAL
 Equals `.TRUE.` if *myproc* collaborates.

firstcl, lastcl INTEGER
 First and last collaborator of *myproc*.
myproc collaborates with:
firstcl, ..., myproc-1, myproc+1, ..., lastcl
 If *myproc* = *firstcl*, there are no collaborators on the left. If *myproc* = *lastcl*, there are no collaborators on the right.
 If *firstcl* = 0 and *lastcl* = *nprocs-1*, then *myproc* collaborates with everybody

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

pmpim2

Computes the eigenpair range assignments for all processes.

Syntax

```
call pmpim2( il, iu, nprocs, pmyils, pmyius )
```

Description

pmpim2 is the scheduling subroutine. It computes for all processors the eigenpair range assignments.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

il, iu INTEGER
 The range of eigenpairs to be computed.

nprocs INTEGER
 The total number of processors available.

Output Parameters

pmyls INTEGER array
 For each processor p , $pmyls(p)$ is the index of the first eigenvalue in a cluster to be computed.
pmyls(p) equals zero if p stays idle.

pmihs INTEGER array
 For each processor p , $pmihs(p)$ is the index of the last eigenvalue in a cluster to be computed.
pmihs(p) equals zero if p stays idle.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
call ccombamax1(v1, v2)
```

```
call zcombamax1(v1, v2)
```

Description

The ?combamax1 routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

v1 (local)
 COMPLEX for ccombamax1
 COMPLEX*16 for zcombamax1
 Array of size 2. The first maximum absolute value element and its global index. $v1(1)=amax$, $v1(2)=indx$.

v2 (local)
 COMPLEX for ccombamax1
 COMPLEX*16 for zcombamax1
 Array of size 2. The second maximum absolute value element and its global index. $v2(1)=amax$, $v2(2)=indx$.

Output Parameters

v1 (local).
 The first maximum absolute value element and its global index.
 $v1(1)=amax$, $v1(2)=indx$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS `p?asum`, but using the true absolute value.

Syntax

```
call pscsum1(n, asum, x, ix, jx, descx, incx)
```

```
call pdzsum1(n, asum, x, ix, jx, descx, incx)
```

Description

The `p?sum1` routine returns the sum of absolute values of a complex distributed vector `sub(x)` in `asum`, where `sub(x)` denotes $X(ix:ix+n-1, jx:jx)$, if `incx = 1`, $X(ix:ix, jx:jx+n-1)$, if `incx = m_x`.

Based on `p?asum` from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

<code>n</code>	(global) INTEGER. The number of components of the distributed vector <code>sub(x)</code> . $n \geq 0$.
<code>x</code>	(local) COMPLEX for <code>pscsum1</code> COMPLEX*16 for <code>pdzsum1</code> . Pointer into the local memory to an array of size $(lld_x, LOCC(jx+n-1))$. This array contains the local pieces of the distributed vector <code>sub(X)</code> .
<code>ix</code>	(global) INTEGER. The row index in the global matrix <code>X</code> indicating the first row of <code>sub(X)</code> .
<code>jx</code>	(global) INTEGER. The column index in the global matrix <code>X</code> indicating the first column of <code>sub(X)</code> .
<code>descx</code>	(local) INTEGER. Array of size <code>dlen_ = 9</code> . The array descriptor for the distributed matrix <code>X</code> .
<code>incx</code>	(global) INTEGER. The global increment for the elements of <code>X</code> . Only two values of <code>incx</code> are supported in this version, namely 1 and <code>m_x</code> .

Output Parameters

<code>asum</code>	(local) Pointer to REAL. The sum of absolute values of the distributed vector <code>sub(X)</code> only in its scope.
-------------------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The routine is called by p?dbtrs.

Syntax

```
call psdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pddbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pcdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pzdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

Description

The p?dbtrsv routine solves a banded triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ (for real flavors); $A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ (for complex flavors),

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code of p?dbtrf and is stored in $A(1:n, ja:ja+n-1)$ and af . The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

The routine p?dbtrf must be called first.

Input Parameters

<i>uplo</i>	(global) CHARACTER. If <i>uplo</i> = 'U', the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if <i>uplo</i> = 'L', the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$, if <i>trans</i> = 'C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix A ; ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
<i>bwu</i>	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
<i>a</i>	(local). REAL for psdbtrsv

	DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbtrsv COMPLEX*16 for pzdbtrsv. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$, where $lld_a \geq (bwl+bwu+1)$. On entry, this array contains the local pieces of the n -by- n unsymmetric banded distributed Cholesky factor L or L^T , represented in global A as $A(1:n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.
<i>ja</i>	(global) INTEGER. The index in the global matrix A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. if 1d type ($dtype_a = 501$ or 502), $dlen \geq 7$; if 2d type ($dtype_a = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
<i>b</i>	(local) REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv COMPLEX for pcdbtrsv COMPLEX*16 for pzdbtrsv. Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right-hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global matrix B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
<i>descb</i>	(global and local) INTEGER array of size $dlen_$. if 1d type ($dtype_b = 502$), $dlen \geq 7$; if 2d type ($dtype_b = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix B . Contains information of mapping B to memory.
<i>laf</i>	(local) INTEGER. Size of user-input auxiliary fill-in space af . $laf \geq nb * (bwl+bwu) + 6 * \max(bwl, bwu) * \max(bwl, bwu)$. If laf is not large enough, an error code is returned and the minimum acceptable size will be returned in $af(1)$.
<i>work</i>	(local). REAL for psdbtrsv DOUBLE PRECISION for pddbtrsv

COMPLEX for p_cdbtrsv
 COMPLEX*16 for p_zdbtrsv.

Temporary workspace. This space may be overwritten in between calls to routines.

work must be the size given in *lwork*.

lwork

(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

$lwork \geq \max(bwl, bwu) * nrhs$.

Output Parameters

a

(local).

This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.

b

On exit, this contains the local piece of the solutions distributed matrix *X*.

af

(local).

REAL for p_sdbtrsv
 DOUBLE PRECISION for p_ddbtrsv
 COMPLEX for p_cdbtrsv
 COMPLEX*16 for p_zdbtrsv.

auxiliary fill-in space. The fill-in space is created in a call to the factorization routine p_?dbtrf and is stored in *af*. If a linear system is to be solved using p_?dbtrf after the factorization routine, *af* must not be altered after the factorization.

work

On exit, *work*(1) contains the minimal *lwork*.

info

(local).

INTEGER. If *info* = 0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p_?dttrsv

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p_?dttrs.

Syntax

```
call psdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pddttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pcdttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

```
call pzdtttrsv(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

Description

The `p?dttrsv` routine solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs) \text{ or}$$

$$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs) \text{ for real flavors; } A(1:n, ja:ja+n-1)^{H*} X = B(ib:ib+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a tridiagonal matrix factor produced by the Gaussian elimination code of `p?dttrf` and is stored in $A(1:n, ja:ja+n-1)$ and af .

The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

The routine `p?dttrf` must be called first.

Input Parameters

<i>uplo</i>	(global) CHARACTER. If $uplo='U'$, the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if $uplo = 'L'$, the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. If $trans = 'N'$, solve with $A(1:n, ja:ja+n-1)$, if $trans = 'C'$, solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix A ; ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix $B(ib:ib+n-1, 1:nrhs)$. ($nrhs \geq 0$).
<i>dl</i>	(local). REAL for <code>psdttrsv</code> DOUBLE PRECISION for <code>pddttrsv</code> COMPLEX for <code>pcdttrsv</code> COMPLEX*16 for <code>pzdtttrsv</code> . Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, $dl(1)$ is not referenced, and dl must be aligned with d .

	Must be of size $\geq nb_a$.
<i>d</i>	<p>(local).</p> <p>REAL for psdttrsv</p> <p>DOUBLE PRECISION for pddttrsv</p> <p>COMPLEX for pcdttrsv</p> <p>COMPLEX*16 for pzdtttrsv.</p> <p>Pointer to local part of global vector storing the main diagonal of the matrix.</p>
<i>du</i>	<p>(local).</p> <p>REAL for psdttrsv</p> <p>DOUBLE PRECISION for pddttrsv</p> <p>COMPLEX for pcdttrsv</p> <p>COMPLEX*16 for pzdtttrsv.</p> <p>Pointer to local part of global vector storing the upper diagonal of the matrix.</p> <p>Globally, $du(n)$ is not referenced, and <i>du</i> must be aligned with <i>d</i>.</p>
<i>ja</i>	(global) INTEGER. The index in the global matrix <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>.</p> <p>if 1<i>d</i> type (<i>dtype_a</i> = 501 or 502), <i>dlen</i> \geq 7;</p> <p>if 2<i>d</i> type (<i>dtype_a</i> = 1), <i>dlen</i> \geq 9.</p> <p>The array descriptor for the distributed matrix <i>A</i>. Contains information of mapping of <i>A</i> to memory.</p>
<i>b</i>	<p>(local)</p> <p>REAL for psdttrsv</p> <p>DOUBLE PRECISION for pddttrsv</p> <p>COMPLEX for pcdttrsv</p> <p>COMPLEX*16 for pzdtttrsv.</p> <p>Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right-hand sides $B(ib:ib+n-1, 1 : nrhs)$.</p>
<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>.</p> <p>if 1<i>d</i> type (<i>dtype_b</i> = 502), <i>dlen</i> \geq 7;</p> <p>if 2<i>d</i> type (<i>dtype_b</i> = 1), <i>dlen</i> \geq 9.</p>

The array descriptor for the distributed matrix B . Contains information of mapping B to memory.

laf

(local).

INTEGER.

Size of user-input auxiliary fill-in space af .

$laf \geq 2 * (nb+2)$. If laf is not large enough, an error code is returned and the minimum acceptable size will be returned in $af(1)$.

work

(local).

REAL for `psdttrsv`DOUBLE PRECISION for `pddttrsv`COMPLEX for `pcdttrsv`COMPLEX*16 for `pzdttrsv`.

Temporary workspace. This space may be overwritten in between calls to routines.

work must be the size given in *lwork*.

lwork

(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work(1)* and an error code is returned.

$lwork \geq 10 * npcol + 4 * nrhs$.

Output Parameters

dl

(local).

On exit, this array contains information containing the factors of the matrix.

d

On exit, this array contains information containing the factors of the matrix. Must be of size $\geq nb_a$.

b

On exit, this contains the local piece of the solutions distributed matrix X .

af

(local).

REAL for `psdttrsv`DOUBLE PRECISION for `pddttrsv`COMPLEX for `pcdttrsv`COMPLEX*16 for `pzdttrsv`.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?dttrf` and is stored in *af*. If a linear system is to be solved using `p?dttrs` after the factorization routine, *af* must not be altered after the factorization.

work

On exit, *work(1)* contains the minimal *lwork*.

info

(local). INTEGER.

If $info=0$, the execution is successful.

if $info < 0$: If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gebal

Balances a general real matrix.

Syntax

```
call psgebal( job, n, a, desca, ilo, ihi, scale, info )
```

```
call pdgebal( job, n, a, desca, ilo, ihi, scale, info )
```

Description

p?gebal balances a general real matrix A . This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to $ilo-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

Input Parameters

<i>job</i>	(global) CHARACTER*1 Specifies the operations to be performed on <i>a</i> : = 'N': none: simply set $ilo = 1, ihi = n, scale(i) = 1.0$ for $i = 1, \dots, n$; = 'P': permute only; = 'S': scale only; = 'B': both permute and scale.
<i>n</i>	(global) INTEGER The order of the matrix A ($n \geq 0$).
<i>a</i>	REAL for psgebal DOUBLE PRECISION for pdgebal (local) Pointer into the local memory to an array of size ($lld_a, LOC_c(n)$) This array contains the local pieces of global input matrix A .
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

OUTPUT Parameters

a On exit, *a* is overwritten by the balanced matrix.

If $job = 'N'$, a is not referenced.

See Notes for further details.

ilo, ihi

(global) INTEGER

ilo and ihi are set to integers such that on exit matrix elements $A(i,j)$ are zero if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $job = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

$scale$

REAL for psgebal

DOUBLE PRECISION for pdgebal

(global) array of size n .

Details of the permutations and scaling factors applied to a . If pj is the index of the row and column interchanged with row and column j and dj is the scaling factor applied to row and column j , then

$scale(j) = pj$ for $j = 1, \dots, ilo-1$

$scale(j) = dj$ for $j = ilo, \dots, ihi$

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

$info$

(global) INTEGER

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

Application Notes

The permutations consist of row and column interchanges which put the matrix in the form

$$PAP = \begin{pmatrix} T_1 & X & Y \\ \mathbf{0} & B & Z \\ \mathbf{0} & \mathbf{0} & T_2 \end{pmatrix}$$

where T_1 and T_2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ilo and ihi mark the starting and ending columns of the submatrix B . Balancing consists of applying a diagonal similarity transformation $D^{-1}BD$ to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

$$\begin{pmatrix} T_1 & XD & Y \\ \mathbf{0} & D^{-1}BD & D^{-1}Z \\ \mathbf{0} & \mathbf{0} & T_2 \end{pmatrix}$$

Information about the permutations P and the diagonal matrix D is returned in the vector $scale$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
call psgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pdgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pcgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

```
call pzgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

Description

The p?gebd2 routine reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q^* \text{sub}(A) P = B.$$

If $m \geq n$, B is the upper bidiagonal; if $m < n$, B is the lower bidiagonal.

Input Parameters

m	(global) INTEGER. The number of rows of the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2 COMPLEX for pcgebd2 COMPLEX*16 for pzgebd2. Pointer into the local memory to an array of size $(lld_a, LOC_c(ja+n-1))$. On entry, this array contains the local pieces of the general distributed matrix $\text{sub}(A)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
$work$	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

This is a workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least $lwork \geq \max(mpa0, nqa0)$,

where $nb = mb_a = nb_a$, $iroffa = \text{mod}(ia-1, nb)$,

$iarow = \text{indxg2p}(ia, nb, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb, mycol, csrc_a, npcot)$,

$mpa0 = \text{numroc}(m+iroffa, nb, myrow, iarow, nprow)$,

$nqa0 = \text{numroc}(n+icoffa, nb, mycol, iacol, npcot)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

(local).

On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix *B*; the elements below the diagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the first superdiagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix *B*; the elements below the first subdiagonal, with the array *tauq*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors, and the elements above the diagonal, with the array *taup*, represent the orthogonal matrix *P* as a product of elementary reflectors. See *Applications Notes* below.

d

(local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix *B*: $d(i) = a(i, i)$. *d* is tied to the distributed matrix *A*.

e

(local)

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-2)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B :

if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$;

if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$. e is tied to the distributed matrix A .

tauq

(local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCc(ja+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . *tauq* is tied to the distributed matrix A .

taup

(local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Array of size $LOCr(ia+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix P . *taup* is tied to the distributed matrix A .

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local)

INTEGER.

If *info* = 0, the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1) * H(2) * \dots * H(n), \text{ and } P = G(1) * G(2) * \dots * G(n-1)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \tau_{uq} * v * v', \text{ and } G(i) = I - \tau_{up} * u * u',$$

where τ_{uq} and τ_{up} are real/complex scalars, and v and u are real/complex vectors. $v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i-ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$.

If $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i+1:ia+m-1, ja+i-1)$;

$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples:

$$\begin{array}{l}
 m = 6 \text{ and } n = 5 (m > n) : \\
 \begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}
 \end{array}
 \qquad
 \begin{array}{l}
 m = 5 \text{ and } n = 6 (m < n) : \\
 \begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}
 \end{array}$$

where d and e denote diagonal and off-diagonal elements of B , v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

call psgehd2(*n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

call pdgehd2(*n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

call pcgehd2(*n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

call pzgehd2(*n*, *ilo*, *ihi*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

Description

The p?gehd2 routine reduces a real/complex general distributed matrix $\text{sub}(A)$ to upper Hessenberg form H by an orthogonal/unitary similarity transformation: $Q' * \text{sub}(A) * Q = H$, where $\text{sub}(A) = A(ia+n-1 : ia+n-1, ja+n-1 : ja+n-1)$.

Input Parameters

n (global) INTEGER. The order of the distributed submatrix A . ($n \geq 0$).

ilo, *ihi* (global) INTEGER. It is assumed that the matrix $\text{sub}(A)$ is already upper triangular in rows $ia:ia+ilo-2$ and $ia+ihi:ia+n-1$ and columns $ja:ja+jlo-2$ and $ja+jhi:ja+n-1$. See *Application Notes* for further information.

If $n \geq 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set $ilo = 1$, $ihi = n$.

a

(local).

REAL for psg ehd2

DOUBLE PRECISION for pdg ehd2

COMPLEX for pcg ehd2

COMPLEX*16 for pzg ehd2.

Pointer into the local memory to an array of size $(lld_a, LOC_c(ja+n-1))$.

On entry, this array contains the local pieces of the n -by- n general distributed matrix $sub(A)$ to be reduced.

ia, ja

(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of $sub(A)$, respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

work

(local).

REAL for psg ehd2

DOUBLE PRECISION for pdg ehd2

COMPLEX for pcg ehd2

COMPLEX*16 for pzg ehd2.

This is a workspace array of size $lwork$.

lwork

(local or global) INTEGER.

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq nb + \max(npa0, nb)$, where $nb = mb_a = nb_a$, $irow = \text{mod}(ia-1, nb)$, $iarow = \text{indxg2p}(ia, nb, myrow, rsrc_a, nprow)$, $npa0 = \text{numroc}(ihi + irow, nb, myrow, iarow, nprow)$.

indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

(local). On exit, the upper triangle and the first subdiagonal of $sub(A)$ are overwritten with the upper Hessenberg matrix H , and the elements below the first subdiagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors. (see *Application Notes* below).

tau

(local).

REAL for psg ehd2

DOUBLE PRECISION for pdgehd2

COMPLEX for pcgehd2

COMPLEX*16 for pzgehd2.

Array of size $LOCc(ja+n-2)$ The scalar factors of the elementary reflectors (see *Application Notes* below). Elements $ja:ja+ilo-2$ and $ja+ihi:ja+n-2$ of the global vector τ are set to zero. τ is tied to the distributed matrix A .

work

On exit, $work(1)$ returns the minimal and optimal $lwork$.

info

(local) INTEGER.

If $info = 0$, the execution is successful.

if $info < 0$: If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo)*H(ilo+1)*...*H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau*v*v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i)=0$, $v(i+1)=1$ and $v(ihi+1:n)=0$; $v(i+2:ihi)$ is stored on exit in $A(ia+ilo+i:ia+ihi-1, ia+ilo+i-2)$, and τ in $\tau(ja+ilo+i-2)$.

The contents of $A(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ & & & & & & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ a & h & h & h & h & h & a \\ h & h & h & h & h & h & h \\ v2 & h & h & h & h & h & h \\ v2 & v3 & h & h & h & h & h \\ v2 & v3 & v4 & h & h & h & h \\ & & & & & & a \end{bmatrix}$

where a denotes an element of the original matrix $\text{sub}(A)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+ilo+i-2)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

call psgelq2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *tau*, *work*, *lwork*, *info*)

```
call pdgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?gelq2` routine computes an LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows of the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns of the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for <code>psgelq2</code> DOUBLE PRECISION for <code>pdgelq2</code> COMPLEX for <code>pcgelq2</code> COMPLEX*16 for <code>pzgelq2</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local). REAL for <code>psgelq2</code> DOUBLE PRECISION for <code>pdgelq2</code> COMPLEX for <code>pcgelq2</code> COMPLEX*16 for <code>pzgelq2</code> . This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where $irow = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprw)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcwl)$, $mp0 = \text{numroc}(m+irow, mb_a, myrow, iarow, nprw)$,

$ng0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npc0l),$

$indxg2p$ and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a	(local). On exit, the elements on and below the diagonal of $\text{sub}(A)$ contain the m by $\min(m,n)$ lower trapezoidal matrix L (L is lower triangular if $m \leq n$); the elements above the diagonal, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
τ	(local). REAL for <code>psgelq2</code> DOUBLE PRECISION for <code>pdgelq2</code> COMPLEX for <code>pcgelq2</code> COMPLEX*16 for <code>pzgelq2</code> . Array of size $LOCr(ia+\min(m, n)-1)$. This array contains the scalar factors of the elementary reflectors. τ is tied to the distributed matrix A .
$work$	On exit, $work(1)$ returns the minimal and optimal $lwork$.
$info$	(local) INTEGER. If $info = 0$, the execution is successful. if $info < 0$: If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia+k-1) * H(ia+k-2) * \dots * H(ia)$ for real flavors, $Q = (H(ia+k-1))^{H*} (H(ia+k-2))^{H} \dots (H(ia))^{H}$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ (for real flavors) or $\text{conjg}(v(i+1:n))$ (for complex flavors) is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and τ in $\tau(ia+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?geql2 routine computes a QL factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q * L$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(A). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(A). ($n \geq 0$).
<i>a</i>	(local). REAL for psgeql2 DOUBLE PRECISION for pdgeql2 COMPLEX for pcgeql2 COMPLEX*16 for pzgeql2. Pointer into the local memory to an array of size ($lld_a, LOC_c(ja+n-1)$). On entry, this array contains the local pieces of the m -by- n distributed matrix sub(A) which is to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of sub(A), respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A.
<i>work</i>	(local). REAL for psgeql2 DOUBLE PRECISION for pdgeql2 COMPLEX for pcgeql2 COMPLEX*16 for pzgeql2. This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$,

where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npc1)$,

$mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow)$,

$nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npc1)$,

indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npc1* can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	<p>(local).</p> <p>On exit,</p> <p>if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n-by-n lower triangular matrix L;</p> <p>if $m \leq n$, the elements on and below the $(n-m)$-th superdiagonal contain the m-by-n lower trapezoidal matrix L; the remaining elements, with the array <i>tau</i>, represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).</p>
<i>tau</i>	<p>(local).</p> <p>REAL for <code>psgeql2</code></p> <p>DOUBLE PRECISION for <code>pdgeql2</code></p> <p>COMPLEX for <code>pcgeql2</code></p> <p>COMPLEX*16 for <code>pzgeql2</code>.</p> <p>Array of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A.</p>
<i>work</i>	<p>On exit, <i>work</i>(1) returns the minimal and optimal <i>lwork</i>.</p>
<i>info</i>	<p>(local). INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful. if <i>info</i> < 0: If the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = - (<i>i</i>*100+<i>j</i>), if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p>

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $\tau(ja+n-k+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pcgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pzgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?geqr2` routine computes a QR factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q^*R$.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for <code>psgeqr2</code> DOUBLE PRECISION for <code>pdgeqr2</code> COMPLEX for <code>pcgeqr2</code> COMPLEX*16 for <code>pzgeqr2</code> . Pointer into the local memory to an array of size $(lld_a, LOC_c(ja+n-1))$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
$work$	(local). REAL for <code>psgeqr2</code> DOUBLE PRECISION for <code>pdgeqr2</code>

COMPLEX for `pcgeqr2`

COMPLEX*16 for `pzgeqr2`.

This is a workspace array of size `lwork`.

`lwork`

(local or global) INTEGER.

The size of the array `work`.

`lwork` is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$,

where $irow = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$,

$mp0 = \text{numroc}(m+irow, mb_a, myrow, iarow, nprow)$,

$nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

`a`

(local).

On exit, the elements on and above the diagonal of `sub(A)` contain the $\min(m,n)$ by n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

`tau`

(local).

REAL for `psgeqr2`

DOUBLE PRECISION for `pdgeqr2`

COMPLEX for `pcgeqr2`

COMPLEX*16 for `pzgeqr2`.

Array of size $LOCc(ja + \min(m,n) - 1)$. This array contains the scalar factors of the elementary reflectors. `tau` is tied to the distributed matrix A .

`work`

On exit, `work(1)` returns the minimal and optimal `lwork`.

`info`

(local) INTEGER.

If `info = 0`, the execution is successful. if `info < 0`:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$,

if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(j) = I - \tau * v * v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
call psgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?gerq2 routine computes an RQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = R * Q$.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix sub(A). ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix sub(A). ($n \geq 0$).
a	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. Pointer into the local memory to an array of size $(lld_a, LOC_c(ja+n-1))$. On entry, this array contains the local pieces of the m -by- n distributed matrix sub(A) which is to be factored.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of sub(A), respectively.

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>work</i>	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npc0l)$, $mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow)$, $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npc0l)$, <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	(local). On exit, if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. Array of size $LOCr(ia+m-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
 If *info* = 0, the execution is successful.
 if *info* < 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1)$ for real flavors,

$Q = (H(ia))^{H*} (H(ia+1))^{H*} \dots (H(ia+k-1))^{H*}$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$,

where τ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ for real flavors or $\text{conjg}(v(1:n-k+i-1))$ for complex flavors is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ in $\tau(ia+m-k+i-1)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

call psgetf2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *info*)

call pdgetf2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *info*)

call pcgetf2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *info*)

call pzgetf2(*m*, *n*, *a*, *ia*, *ja*, *desca*, *ipiv*, *info*)

Description

The *p?getf2* routine computes an *LU* factorization of a general *m*-by-*n* distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using partial pivoting with row interchanges.

The factorization has the form $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-

Optimization Notice

dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(<i>A</i>). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>A</i>). ($nb_a - \text{mod}(ja-1, nb_a) \geq n \geq 0$).
<i>a</i>	(local). REAL for <code>psgetf2</code> DOUBLE PRECISION for <code>pdgetf2</code> COMPLEX for <code>pcgetf2</code> COMPLEX*16 for <code>pzgetf2</code> . Pointer into the local memory to an array of size ($lld_a, LOC_c(ja+n-1)$). On entry, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix sub(<i>A</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>ipiv</i>	(local) INTEGER. Array of size ($LOC_r(m_a) + mb_a$). This array contains the pivoting information. <i>ipiv</i> (<i>i</i>) -> The global row that local row <i>i</i> was swapped with. This array is tied to the distributed matrix <i>A</i> .
<i>info</i>	(local). INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> < 0: <ul style="list-style-type: none"> • if the <i>i</i>-th argument is an array and the <i>j</i>-th entry had an illegal value, then <i>info</i> = $-(i*100+j)$, • if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

If $info > 0$: If $info = k$, the matrix element $U_{(ia+k-1, ja+k-1)}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

```
call pslabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

```
call pdlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

```
call pclabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

```
call pzlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy, jy, descy, work)
```

Description

The p?labrd routine reduces the first nb rows and columns of a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q^* A P$, and returns the matrices X and Y necessary to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $m \geq n$, $\text{sub}(A)$ is reduced to upper bidiagonal form; if $m < n$, $\text{sub}(A)$ is reduced to lower bidiagonal form.

This is an auxiliary routine called by p?gebrd.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
nb	(global) INTEGER. The number of leading rows and columns of $\text{sub}(A)$ to be reduced.
a	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd

COMPLEX*16 for pzlabrd.

Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$.

On entry, this array contains the local pieces of the general distributed matrix $sub(A)$.

ia, ja

(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $sub(A)$, respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

ix, jx

(global) INTEGER. The row and column indices in the global matrix X indicating the first row and the first column of the matrix $sub(X)$, respectively.

descx

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix X .

iy, jy

(global) INTEGER. The row and column indices in the global matrix Y indicating the first row and the first column of the matrix $sub(Y)$, respectively.

descy

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Y .

work

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Workspace array of size $lwork$.

$lwork \geq nb_a + nq$,

with $nq = \text{numroc}(n + \text{mod}(ia-1, nb_y), nb_y, mycol, iacol, npcol)$

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$

indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

a

(local)

On exit, the first nb rows and columns of the matrix are overwritten; the rest of the distributed matrix $sub(A)$ is unchanged.

If $m \geq n$, elements on and below the diagonal in the first nb columns, with the array $tauq$, represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array $taup$, represent the orthogonal/unitary matrix P as a product of elementary reflectors.

If $m < n$, elements below the diagonal in the first nb columns, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first nb rows, with the array *taup*, represent the orthogonal/unitary matrix P as a product of elementary reflectors. See *Application Notes* below.

d

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array of size $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal distributed matrix B :

$$d(i) = A(ia+i-1, ja+i-1).$$

d is tied to the distributed matrix A .

e

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array of size $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B :

$$\text{if } m \geq n, E(i) = A(ia+i-1, ja+i) \text{ for } i = 1, 2, \dots, n-1;$$

$$\text{if } m < n, E(i) = A(ia+i, ja+i-1) \text{ for } i = 1, 2, \dots, m-1.$$

e is tied to the distributed matrix A .

tauq, taup

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Array size $LOCc(ja+\min(m,n)-1)$ for *tauq*, size $LOCr(ia+\min(m,n)-1)$ for *taup*. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q for *tauq*, P for *taup*. *tauq* and *taup* are tied to the distributed matrix A . See *Application Notes* below.

x

(local)

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of size lld_x by nb . On exit, the local pieces of the distributed m -by- nb matrix $X(ix:ix+m-1, jx:jx+nb-1)$ required to update the unreduced part of $sub(A)$.

 y

(local).

REAL for pslabrd

DOUBLE PRECISION for pdlabrd

COMPLEX for pclabrd

COMPLEX*16 for pzlabrd

Pointer into the local memory to an array of size lld_y by nb . On exit, the local pieces of the distributed n -by- nb matrix $Y(iy:iy+n-1, jy:jy+nb-1)$ required to update the unreduced part of $sub(A)$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

$$Q = H(1)*H(2)*\dots*H(nb), \text{ and } P = G(1)*G(2)*\dots*G(nb)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \tau_{uq} * v * v', \text{ and } G(i) = I - \tau_{up} * u * u',$$

where τ_{uq} and τ_{up} are real/complex scalars, and v and u are real/complex vectors.

If $m \geq n$, $v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in

$A(ia+i-1:ia+m-1, ja+i-1)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$.

If $m < n$, $v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i+1:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; τ_{uq} is stored in $\tau_{uq}(ja+i-1)$ and τ_{up} in $\tau_{up}(ia+i-1)$. The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are necessary, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $sub(A) := sub(A) - V*Y' - X*U'$. The contents of $sub(A)$ on exit are illustrated by the following examples with $nb = 2$:

$m = 6$ and $n = 5 (m > n)$:

$$\begin{bmatrix} 1 & 1 & u1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
call pslacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

Description

The p?lacon routine estimates the 1-norm of a square, real/unitary distributed matrix A . Reverse communication is used for evaluating matrix-vector products. x and v are aligned with the distributed matrix A , this information is implicitly contained within iv , ix , $descv$, and $descx$.

Input Parameters

n	(global) INTEGER. The length of the distributed vectors v and x . $n \geq 0$.
v	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of size $LOCr(n+\text{mod}(iv-1, mb_v))$. On the final return, $v = a*w$, where $est = \text{norm}(v)/\text{norm}(w)$ (w is not returned).
iv, jv	(global) INTEGER. The row and column indices in the global matrix V indicating the first row and the first column of the submatrix V , respectively.
$descv$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V .
x	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of size $LOCr(n+\text{mod}(ix-1, mb_x))$.
ix, jx	(global) INTEGER. The row and column indices in the global matrix X indicating the first row and the first column of the submatrix X , respectively.

<i>descx</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix X.
<i>isgn</i>	(local). INTEGER. Array of size $LOCr(n+\text{mod}(ix-1, mb_x))$. <i>isgn</i> is aligned with <i>x</i> and <i>v</i> .
<i>kase</i>	(local). INTEGER. On the initial call to <code>p?lacon</code> , <i>kase</i> should be 0.

Output Parameters

<i>x</i>	(local). On an intermediate return, X should be overwritten by A^*X , if <i>kase</i> =1, A'^*X , if <i>kase</i> =2, <code>p?lacon</code> must be re-called with all the other parameters unchanged.
<i>est</i>	(global). REAL for single precision flavors DOUBLE PRECISION for double precision flavors
<i>kase</i>	(local) INTEGER. On an intermediate return, <i>kase</i> is 1 or 2, indicating whether X should be overwritten by A^*X , or A'^*X . On the final return from <code>p?lacon</code> , <i>kase</i> is again 0.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

```
call pslaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

```
call pdlaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

Description

The `p?laconsb` routine looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift QR iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

<i>a</i>	(global). REAL for <code>pslaconsb</code> DOUBLE PRECISION for <code>pdlaconsb</code> Array of size $(lld_a,*)$. On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.

<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER. The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>h44, h33, h43h34</i>	(global). REAL for <code>pslaconsb</code> DOUBLE PRECISION for <code>pdlaconsb</code> These three values are for the double shift QR iteration.
<i>lwork</i>	(local or global) INTEGER. This must be at least $7 * \text{ceil}(\text{ceil}((i-1)/hbl) / \text{lcm}(nprow, npcold))$. Here <code>lcm</code> is least common multiple and $nprow * npcold$ is the logical grid size.

Output Parameters

<i>m</i>	(global). On exit, this yields the starting location of the QR double shift. This will satisfy: $i \leq m \leq i-2$.
<i>buf</i>	(local). REAL for <code>pslaconsb</code> DOUBLE PRECISION for <code>pdlaconsb</code> Array of size <i>lwork</i> .
<i>lwork</i>	(local or global). On exit, <i>lwork</i> is the size of the work buffer.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
call pslacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

The `p?lacp2` routine copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, `p?lacp2` performs a local copy $\text{sub}(A) := \text{sub}(B)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, a:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

`p?lacp2` requires that only dimension of the matrix operands is distributed.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the part of the distributed matrix sub(<i>A</i>) to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of sub(<i>A</i>) is not referenced; = 'L': Lower triangular part is copied; the strictly upper triangular part of sub(<i>A</i>) is not referenced. Otherwise: all of the matrix sub(<i>A</i>) is copied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(<i>A</i>). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>A</i>). ($n \geq 0$).
<i>a</i>	(local). REAL for pslacp2 DOUBLE PRECISION for pdlacp2 COMPLEX for pclacp2 COMPLEX*16 for pzlacp2. Pointer into the local memory to an array of size (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix sub(<i>A</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_a</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_b</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	(local). REAL for pslacp2 DOUBLE PRECISION for pdlacp2 COMPLEX for pclacp2 COMPLEX*16 for pzlacp2. Pointer into the local memory to an array of size (<i>lld_b</i> , <i>LOCc(jb+n-1)</i>). This array contains on exit the local pieces of the distributed matrix sub(<i>B</i>) set as follows:
----------	---

if $uplo = 'U'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j$, $1 \leq j \leq n$;
 if $uplo = 'L'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $j \leq i \leq m$, $1 \leq j \leq n$;
 otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m$, $1 \leq j \leq n$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
call pslacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pdlacp3(m, i, a, desca, b, ldb, ii, jj, rev)
```

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vice versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>m</i>	(global) INTEGER. <i>m</i> is the order of the square submatrix that is copied. $m \geq 0$. Unchanged on exit.
<i>i</i>	(global) INTEGER. $A(i, i)$ is the global location that the copying starts from. Unchanged on exit.
<i>a</i>	(global). REAL for pslacp3 DOUBLE PRECISION for pdlacp3 Array of size (<i>lld_a</i> , *). On entry, the parallel matrix to be copied into or from.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local).

REAL for pslacp3

DOUBLE PRECISION for pdlacp3

Array of size ldb by m . If $rev = 0$, this is the global portion of the array $A(i:i+m-1, i:i+m-1)$. If $rev = 1$, this is unchanged on exit.

ldb

(local)

INTEGER.

The leading dimension of B .

ii

(global) INTEGER. By using rev 0 and 1, data can be sent out and returned again. If $rev = 0$, then ii is destination row index for the node(s) receiving the replicated B . If $ii \geq 0, jj \geq 0$, then node (ii, jj) receives the data. If $ii = -1, jj \geq 0$, then all rows in column jj receive the data. If $ii \geq 0, jj = -1$, then all cols in row ii receive the data. If $ii = -1, jj = -1$, then all nodes receive the data. If $rev \neq 0$, then ii is the source row index for the node(s) sending the replicated B .

jj

(global) INTEGER. Similar description as ii above.

rev

(global) INTEGER. Use $rev = 0$ to send global A into locally replicated B (on node (ii, jj)). Use $rev \neq 0$ to send locally replicated B from node (ii, jj) to its owner (which changes depending on its location in A) into the global A .

Output Parameters

a

(global). On exit, if $rev = 1$, the copied data. Unchanged on exit if $rev = 0$.

b

(local). If $rev = 1$, this is unchanged on exit.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call pslacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

```
call pdlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

```
call pclacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

```
call pzlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

Description

The p?lacpy routine copies all or part of a distributed matrix A to another distributed matrix B . No communication is performed, p?lacpy performs a local copy $\text{sub}(B) := \text{sub}(A)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

Input Parameters

<i>uplo</i>	<p>(global) CHARACTER. Specifies the part of the distributed matrix sub(A) to be copied:</p> <p>= 'U': Upper triangular part; the strictly lower triangular part of sub(A) is not referenced;</p> <p>= 'L': Lower triangular part; the strictly upper triangular part of sub(A) is not referenced.</p> <p>Otherwise: all of the matrix sub(A) is copied.</p>
<i>m</i>	<p>(global) INTEGER.</p> <p>The number of rows in the distributed matrix sub(A). ($m \geq 0$).</p>
<i>n</i>	<p>(global) INTEGER.</p> <p>The number of columns in the distributed matrix sub(A). ($n \geq 0$).</p>
<i>a</i>	<p>(local).</p> <p>REAL for pslacpy</p> <p>DOUBLE PRECISION for pdlacpy</p> <p>COMPLEX for pclacpy</p> <p>COMPLEX*16 for pzlacpy.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$.</p> <p>On entry, this array contains the local pieces of the distributed matrix sub(A).</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix A.</p>
<i>ib, jb</i>	<p>(global) INTEGER. The row and column indices in the global matrix B indicating the first row and the first column of sub(B) respectively.</p>
<i>descb</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix A.</p>

Output Parameters

<i>b</i>	<p>(local).</p> <p>REAL for pslacpy</p> <p>DOUBLE PRECISION for pdlacpy</p> <p>COMPLEX for pclacpy</p> <p>COMPLEX*16 for pzlacpy.</p> <p>Pointer into the local memory to an array of size $(lld_b, LOCc(jb+n-1))$. This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:</p>
----------	---

if $uplo = 'U'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j$, $1 \leq j \leq n$;
 if $uplo = 'L'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $j \leq i \leq m$, $1 \leq j \leq n$;
 otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m$, $1 \leq j \leq n$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
call pslaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pdlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pclaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pzlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
```

Description

The p?laevswp routine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global) INTEGER.

The order of the matrix A . $n \geq 0$.

zin (local).

REAL for pslaevswp

DOUBLE PRECISION for pdlaevswp

COMPLEX for pclaevswp

COMPLEX*16 for pzlaevswp.

Array of size $(ldzi, nvs(iam+2))$. The eigenvectors on input. iam is a process rank from $[0, nprocs)$ interval. Each eigenvector resides entirely in one process. Each process holds a contiguous set of $nvs(iam+2)$ eigenvectors. The global number of the first eigenvector that the process holds is: $((\text{sum for } i=[1, iam+1] \text{ of } nvs(i))+1)$.

$ldzi$ (local)

INTEGER. The leading dimension of the zin array.

iz, jz (global) INTEGER. The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

$descz$ (global and local) INTEGER

	Array of size <i>dlen_</i> . The array descriptor for the distributed matrix Z.
<i>nvs</i>	(global) INTEGER. Array of size <i>nprocs</i> +1 <i>nvs</i> (<i>i</i>) = number of eigenvectors held by processes [0, <i>i</i> -1] <i>nvs</i> (1) = number of eigenvectors held by processes [0, 1 -1) = 0 <i>nvs</i> (<i>nprocs</i> +1)= number of eigenvectors held by processes [0, <i>nprocs</i>)= total number of eigenvectors.
<i>key</i>	(global) INTEGER. Array of size <i>n</i> . Indicates the actual index (after sorting) for each of the eigenvectors.
<i>rwork</i>	(local). REAL for <i>pslaevswp</i> DOUBLE PRECISION for <i>pdlaevswp</i> COMPLEX for <i>pclaevswp</i> COMPLEX*16 for <i>pzlaevswp</i> . Array of size <i>lrwork</i> .
<i>lrwork</i>	(local) INTEGER. Size of <i>work</i> .

Output Parameters

<i>z</i>	(local). REAL for <i>pslaevswp</i> DOUBLE PRECISION for <i>pdlaevswp</i> COMPLEX for <i>pclaevswp</i> COMPLEX*16 for <i>pzlaevswp</i> . Array of global size <i>n</i> by <i>n</i> and of local size (<i>dlen_</i> , <i>nq</i>). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of <i>nb</i> .
----------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lahrd

*Reduces the first *nb* columns of a general rectangular matrix *A* so that elements below the *k*-th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of *A*.*

Syntax

```
call pslahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

```
call pdlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

```
call pclahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

Description

The `p?lahrd` routine reduces the first nb columns of a real general n -by- $(n-k+1)$ distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$ so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q^*A*Q . The routine returns the matrices V and T which determine Q as a block reflector $I-V^*T^*V'$, and also the matrix $Y = A^*V^*T$.

This is an auxiliary routine called by `p?gehrd`. In the following comments $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

n	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
k	(global) INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
nb	(global) INTEGER. The number of columns to be reduced.
a	(local). REAL for <code>pslahrd</code> DOUBLE PRECISION for <code>pdlahrd</code> COMPLEX for <code>pclahrd</code> COMPLEX*16 for <code>pzlahrd</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-k))$. On entry, this array contains the local pieces of the n -by- $(n-k+1)$ general distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
iy, jy	(global) INTEGER. The row and column indices in the global matrix Y indicating the first row and the first column of the matrix $\text{sub}(Y)$, respectively.
$descy$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Y .
$work$	(local). REAL for <code>pslahrd</code> DOUBLE PRECISION for <code>pdlahrd</code>

COMPLEX for pclahrd
 COMPLEX*16 for pzlahrd.
 Array of size nb .

Output Parameters

a	<p>(local).</p> <p>On exit, the elements on and above the k-th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the k-th subdiagonal, with the array τ, represent the matrix Q as a product of elementary reflectors. The other columns of $A(ia:ia+n-1, ja:ja+n-k)$ are unchanged. (See <i>Application Notes</i> below.)</p>
τ	<p>(local)</p> <p>REAL for pslahrd DOUBLE PRECISION for pdlahrd COMPLEX for pclahrd COMPLEX*16 for pzlahrd.</p> <p>Array of size $LOCc(ja+n-2)$. The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). τ is tied to the distributed matrix A.</p>
t	<p>(local) REAL for pslahrd DOUBLE PRECISION for pdlahrd COMPLEX for pclahrd COMPLEX*16 for pzlahrd.</p> <p>Array of size $nb_aby\ nb_a$. The upper triangular matrix T.</p>
y	<p>(local).</p> <p>REAL for pslahrd DOUBLE PRECISION for pdlahrd COMPLEX for pclahrd COMPLEX*16 for pzlahrd.</p> <p>Pointer into the local memory to an array of size $lld_yby\ nb_a$. On exit, this array contains the local pieces of the n-by-nb distributed matrix Y. $lld_y \geq LOCr(ia+n-1)$.</p>

Application Notes

The matrix Q is represented as a product of nb elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $A(ia+i+k:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form: $A(ia:ia+n-1, ja:ja+n-k) := (I-V*Y*V^T)*(A(ia:ia+n-1, ja:ja+n-k)-Y*V^T)$. The contents of $A(ia:ia+n-1, ja:ja+n-k)$ on exit are illustrated by the following example with $n = 7$, $k = 3$, and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix $A(ia:ia+n-1, ja:ja+n-k)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

```
void pslaiect(float *sigma, int *n, float *d, int *count);
void pdlaiectb(float *sigma, int *n, float *d, int *count);
void pdlaiectl(float *sigma, int *n, float *d, int *count);
```

Description

The `p?laiect` routine computes the number of negative eigenvalues of $(A - \sigma I)$. This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine `pslaiect` is assumed to be bit 32. Double-precision routines `pdlaiectb` and `pdlaiectl` differ in the order of the double precision word storage and, consequently, in the signbit location. For `pdlaiectb`, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For `pdlaiectl`, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

<i>sigma</i>	REAL for <code>pslaiect</code> DOUBLE PRECISION for <code>pdlaiectb</code> / <code>pdlaiectl</code> . The shift. <code>p?laiect</code> finds the number of eigenvalues less than equal to <i>sigma</i> .
<i>n</i>	INTEGER. The order of the tridiagonal matrix T . $n \geq 1$.

d REAL for `pslaiect`
 DOUBLE PRECISION for `pdlaiectb/pdlaiectl`.
 Array of size $2n-1$.
 On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix T . These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of T are in the entries $d(1), d(3), \dots, d(2n-1)$, while the squares of the off-diagonal entries are $d(2), d(4), \dots, d(2n-2)$. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $\text{overflow}^{(1/2)} * \text{underflow}^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n INTEGER. The count of the number of eigenvalues of T less than or equal to σ .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lamve

Copies all or part of one two-dimensional distributed array to another.

Syntax

```
call pslamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
call pdlamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
```

Description

`p?lamve` copies all or part of a distributed matrix A to another distributed matrix B . There is no alignment assumptions at all except that A and B are of the same size.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

uplo (global) CHARACTER*1
 Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied:
 = 'U': Upper triangular part is copied; the strictly lower triangular part of $\text{sub}(A)$ is not referenced;

= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced;

Otherwise: All of the matrix sub(*A*) is copied.

<i>m</i>	(global) INTEGER The number of rows to be operated on, which is the number of rows of the distributed matrix sub(<i>A</i>). $m \geq 0$.
<i>n</i>	(global) INTEGER The number of columns to be operated on, which is the number of columns of the distributed matrix sub(<i>A</i>). $n \geq 0$.
<i>a</i>	REAL for pslamve DOUBLE PRECISION for pdlamve (local) pointer into the local memory to an array of size $(lld_a, LOC_c(ja + n - 1))$. This array contains the local pieces of the distributed matrix sub(<i>A</i>) to be copied from.
<i>ia</i>	(global) INTEGER The row index in the global matrix <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER The column index in the global matrix <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib</i>	(global) INTEGER The row index in the global matrix <i>B</i> indicating the first row of sub(<i>B</i>).
<i>jb</i>	(global) INTEGER The column index in the global matrix <i>B</i> indicating the first column of sub(<i>B</i>).
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>dwork</i>	REAL for pslamve DOUBLE PRECISION for pdlamve (local workspace) array If <i>uplo</i> = 'U' or <i>uplo</i> = 'L' and number of processors > 1, the length of <i>dwork</i> is at least as large as the length of <i>b</i> . Otherwise, <i>dwork</i> is not referenced.

OUTPUT Parameters

b REAL for pslamve

DOUBLE PRECISION for pdlamve

(local) pointer into the local memory to an array of size $(lld_b, LOC_c(jb + n - 1))$. This array contains on exit the local pieces of the distributed matrix sub(B).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

`val = pslange(norm, m, n, a, ia, ja, desca, work)`

`val = pdlange(norm, m, n, a, ia, ja, desca, work)`

`val = pclange(norm, m, n, a, ia, ja, desca, work)`

`val = pzlange(norm, m, n, a, ia, ja, desca, work)`

Description

The p?langeroutine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $sub(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) CHARACTER. Specifies what value is returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A, it s not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(A). When $m = 0$, p?lange is set to zero. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(A). When $n = 0$, p?lange is set to zero. $n \geq 0$.
<i>a</i>	(local). REAL for pslange DOUBLE PRECISION for pdlange COMPLEX for pclange

COMPLEX*16 for pzlange.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$ containing the local pieces of the distributed matrix sub(A).

ia, ja

(global) INTEGER. The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(A), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

work

(local).

REAL for pslange

DOUBLE PRECISION for pdlange

COMPLEX for pclange

COMPLEX*16 for pzlange.

Array size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

nq0 if *norm* = '1', 'O' or 'o',

mp0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nrow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *ncol*),

mp0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nrow*),

nq0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *ncol*),

indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *mycol*, *nrow*, and *ncol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val

REAL for pslange/pclange

DOUBLE PRECISION for pdlange/pzlange

The value returned by the routine.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

`val = pslanhs(norm, n, a, ia, ja, desca, work)`

`val = pdlanhs(norm, n, a, ia, ja, desca, work)`

`val = pclanhs(norm, n, a, ia, ja, desca, work)`

`val = pzlanhs(norm, n, a, ia, ja, desca, work)`

Description

The `p?lanhs` routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<code>norm</code>	<p>CHARACTER*1. Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<code>n</code>	<p>(global) INTEGER.</p> <p>The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, <code>p?lanhs</code> is set to zero. $n \geq 0$.</p>
<code>a</code>	<p>(local).</p> <p>REAL for <code>pslanhs</code> DOUBLE PRECISION for <code>pdlanhs</code> COMPLEX for <code>pcplanhs</code> COMPLEX*16 for <code>pzplanhs</code>.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$ containing the local pieces of the distributed matrix $\text{sub}(A)$.</p>
<code>ia, ja</code>	<p>(global) INTEGER.</p> <p>The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.</p>
<code>desca</code>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A.</p>
<code>work</code>	<p>(local).</p> <p>REAL for <code>pslanhs</code> DOUBLE PRECISION for <code>pdlanhs</code> COMPLEX for <code>pcplanhs</code></p>

COMPLEX*16 for pzlanh.

Array of size *lwork*.

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

nq0 if *norm* = '1', 'O' or 'o',

mp0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nrow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *ncol*),

mp0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *nrow*),

nq0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *ncol*),

indxg2p and *numroc* are ScaLAPACK tool routines; *myrow*, *mycol*, *nrow*, and *ncol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

val = pslansy(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pdlansy(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pclansy(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pzlansy(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pplanhe(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

val = pzlanhe(*norm*, *uplo*, *n*, *a*, *ia*, *ja*, *desca*, *work*)

Description

The p?lansy and p?lanheroutines return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$.

Input Parameters

norm (global) CHARACTER. Specifies what value is returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A , it is not a matrix norm.
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
 = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

uplo (global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced.
 = 'U': Upper triangular part of $\text{sub}(A)$ is referenced,
 = 'L': Lower triangular part of $\text{sub}(A)$ is referenced.

n (global) INTEGER.
 The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, *pslansy* is set to zero. $n \geq 0$.

a (local).
 REAL for *pslansy*
 DOUBLE PRECISION for *pdlansy*
 COMPLEX for *pclansy*, *pclanhe*
 COMPLEX*16 for *pzlansy*, *pzlanhe*.
 Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$ containing the local pieces of the distributed matrix $\text{sub}(A)$.
 If *uplo* = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If *uplo* = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja (global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix A .

work (local).
 REAL for *pslansy*, *pclansy*, *pclanhe*
 DOUBLE PRECISION for *pdlansy*, *pzlansy*, *pzlanhe*
 Array of size *lwork*.
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
 $2 * nq0 + mp0 + ldw$ if *norm* = '1', 'O' or 'o', 'I' or 'i',
 where *ldw* is given by:

```

if( nprow#npcol ) then
  ldw = mb_a*iceil(iceil(np0,mb_a), (lcm/nprow))
else
  ldw = 0
end if
0 if norm = 'F', 'f', 'E' or 'e' (not referenced),
where lcm is the least common multiple of nprow and npcold, lcm =
ilcm( nprow, npcold ) and iceil(x,y) is a ScaLAPACK function that
returns ceiling (x/y).
iroffa = mod(ia-1, mb_a ), icoffa = mod( ja-1, nb_a),
iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),
iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcold),
mp0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nq0 = numroc(n+icoffa, nb_a, mycol, iacol, npcold),
ilcm, iceil, indxg2p, and numroc are ScaLAPACK tool functions; myrow,
mycol, nprow, and npcold can be determined by calling the subroutine
blacs_gridinfo.

```

Output Parameters

val The value returned by the routine.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
val = pslantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

```
val = pdlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

```
val = pclantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

```
val = pzlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
```

Description

The p?lantr routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

norm (global) CHARACTER. Specifies what value is returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A , it is not a matrix norm.
 = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
 = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

uplo

(global) CHARACTER.
 Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced.
 = 'U': Upper trapezoidal,
 = 'L': Lower trapezoidal.
 Note that $\text{sub}(A)$ is triangular instead of trapezoidal if $m = n$.

diag

(global) CHARACTER.
 Specifies whether the distributed matrix $\text{sub}(A)$ has unit diagonal.
 = 'N': Non-unit diagonal.
 = 'U': Unit diagonal.

m

(global) INTEGER.
 The number of rows in the distributed matrix $\text{sub}(A)$. When $m = 0$, $p?$ `lantr` is set to zero. $m \geq 0$.

n

(global) INTEGER.
 The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, $p?$ `lantr` is set to zero. $n \geq 0$.

a

(local).
 REAL for `pslantr`
 DOUBLE PRECISION for `pdlantr`
 COMPLEX for `pclantr`
 COMPLEX*16 for `pzlantr`.
 Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$ containing the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja

(global) INTEGER.
 The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

work

(local).
 REAL for `pslantr`

DOUBLE PRECISION for pdlantr
 COMPLEX for pclantr
 COMPLEX*16 for pzlantr.

Array size *lwork*.
lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),
nq0 if *norm* = '1', 'O' or 'o',
mp0 if *norm* = 'I' or 'i',
 0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),
iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),
iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *npro*),
iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),
mp0 = numroc(*m*+*iroffa*, *mb_a*, *myrow*, *iarow*, *npro*),
nq0 = numroc(*n*+*icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),
 indxg2p and numroc are ScaLAPACK tool functions; *myrow*, *mycol*, *npro*,
 and *npcol* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val The value returned by the routine.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

```
call pslapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
            iwork)
```

```
call pdlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
            iwork)
```

```
call pclapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
            iwork)
```

```
call pzlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
            iwork)
```

Description

The `p?lapiv` routine applies either P (permutation matrix indicated by *ipiv*) or $\text{inv}(P)$ to a general m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$, resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix A . This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of $\text{sub}(A)$, pass `rowcol='C'` and `pivroc='C'`.

Input Parameters

<i>direc</i>	(global) CHARACTER*1. Specifies in which order the permutation is applied: = 'F' (Forward): Applies pivots forward from top of matrix. Computes $P \cdot \text{sub}(A)$. = 'B' (Backward): Applies pivots backward from bottom of matrix. Computes $\text{inv}(P) \cdot \text{sub}(A)$.
<i>rowcol</i>	(global) CHARACTER*1. Specifies if the rows or columns are to be permuted: = 'R': Rows will be permuted, = 'C': Columns will be permuted.
<i>pivroc</i>	(global) CHARACTER*1. Specifies whether <i>ipiv</i> is distributed over a process row or column: = 'R': <i>ipiv</i> is distributed over a process row, = 'C': <i>ipiv</i> is distributed over a process column.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$. When $m = 0$, $p?$ <i>lapiv</i> is set to zero. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, $p?$ <i>lapiv</i> is set to zero. $n \geq 0$.
<i>a</i>	(local). REAL for <i>pslapiv</i> DOUBLE PRECISION for <i>pdlapiv</i> COMPLEX for <i>pclapiv</i> COMPLEX*16 for <i>pzlapiv</i> . Pointer into the local memory to an array of size (<i>lld_a</i> , $LOC(ja+n-1)$) containing the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ipiv</i>	(local) INTEGER. Array of size <i>lipiv</i> ; when <i>rowcol</i> ='R' or 'r': $lipiv \geq LOCr(ia+m-1) + mb_a$ if <i>pivroc</i> ='C' or 'c',

$lipiv \geq LOCc(m + \text{mod}(jp-1, nb_p))$ if $pivroc='R'$ or $'r'$, and,
when $rowcol='C'$ or $'c'$:

$lipiv \geq LOCr(n + \text{mod}(ip-1, mb_p))$ if $pivroc='C'$ or $'c'$,
 $lipiv \geq LOCc(ja+n-1) + nb_a$ if $pivroc='R'$ or $'r'$.

This array contains the pivoting information. $ipiv(i)$ is the global row (column), local row (column) i was swapped with. When $rowcol='R'$ or $'r'$ and $pivroc='C'$ or $'c'$, or $rowcol='C'$ or $'c'$ and $pivroc='R'$ or $'r'$, the last piece of this array of size mb_a (resp. nb_a) is used as workspace. In those cases, this array is tied to the distributed matrix A .

ip, jp

(global) INTEGER. The row and column indices in the global matrix P indicating the first row and the first column of the matrix $\text{sub}(P)$, respectively.

$descip$

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed vector $ipiv$.

$iwork$

(local). INTEGER.

Array of size ldw , where ldw is equal to the workspace necessary for transposition, and the storage of the transposed $ipiv$:

Let lcm be the least common multiple of $nrow$ and $ncol$.

```

if( rowcol.eq.'r' .and. pivroc.eq.'r') then
  if( nrow.eq. ncol) then
    ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
  else
    ldw = LOCr( n_p + mod(jp-1, nb_p) ) +
    nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/ncol) )
  end if
else if( rowcol.eq.'c' .and. pivroc.eq.'c') then
  if( nrow.eq. ncol ) then
    ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
  else
    ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
    mb_p * ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nrow) )
  end if
else
  iwork is not referenced.
end if

```

Output Parameters

a

(local).

On exit, the local pieces of the permuted distributed submatrix.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

[p?laqge](#)

Scales a general rectangular matrix, using row and column scaling factors computed by [p?geequ](#).

Syntax

```
call pslaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

Description

The `p?laqge` routine equilibrates a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using the row and scaling factors in the vectors r and c computed by `p?geequ`.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). REAL for <code>pslaqge</code> DOUBLE PRECISION for <code>pdlaqge</code> COMPLEX for <code>pclaqge</code> COMPLEX*16 for <code>pzlaqge</code> . Pointer into the local memory to an array of size $(lld_a, LOcc(ja+n-1))$. On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
r	(local). REAL for <code>pslaqge</code> DOUBLE PRECISION for <code>pdlaqge</code> COMPLEX for <code>pclaqge</code> COMPLEX*16 for <code>pzlaqge</code> . Array of size $LOCr(m_a)$. The row scale factors for $\text{sub}(A)$. r is aligned with the distributed matrix A , and replicated across every process column. r is tied to the distributed matrix A .
c	(local). REAL for <code>pslaqge</code> DOUBLE PRECISION for <code>pdlaqge</code>

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Array of size $LOCc(n_a)$. The row scale factors for sub(A). c is aligned with the distributed matrix A , and replicated across every process column. c is tied to the distributed matrix A .

rowcnd

(local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

The global ratio of the smallest $r(i)$ to the largest $r(i)$, $ia \leq i \leq ia+m-1$.

colcnd

(local).

REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

The global ratio of the smallest $c(i)$ to the largest $c(i)$, $ia \leq i \leq ia+n-1$.

amax

(global). REAL for pslaqge

DOUBLE PRECISION for pdlaqge

COMPLEX for pclaqge

COMPLEX*16 for pzlaqge.

Absolute value of largest distributed submatrix entry.

Output Parameters

a

(local).

On exit, the equilibrated distributed matrix. See *equed* for the form of the equilibrated distributed submatrix.

equed

(global) CHARACTER.

Specifies the form of equilibration that was done.

= 'N': No equilibration

= 'R': Row equilibration, that is, sub(A) has been pre-multiplied by $\text{diag}(r(ia:ia+m-1))$,

= 'C': column equilibration, that is, sub(A) has been post-multiplied by $\text{diag}(c(ja:ja+n-1))$,

= 'B': Both row and column equilibration, that is, sub(A) has been replaced by $\text{diag}(r(ia:ia+m-1)) * \text{sub}(A) * \text{diag}(c(ja:ja+n-1))$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr0

Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.

Syntax

```
call pslaqr0( wantt, wantz, n, ilo, ihi, h, desch, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, liwork, info, relevel )
```

```
call pdlaqr0( wantt, wantz, n, ilo, ihi, h, desch, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, liwork, info, relevel )
```

Description

p?laqr0 computes the eigenvalues of a Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition $H = Z^*T^*Z^T$, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q : $A = Q * H * Q^T = (QZ) * T * (QZ)^T$.

Input Parameters

<i>wantt</i>	(global) LOGICAL = .TRUE. : the full Schur form T is required; = .FALSE. : only eigenvalues are required.
<i>wantz</i>	(global) LOGICAL = .TRUE. : the matrix of Schur vectors Z is required; = .FALSE.: Schur vectors are not required.
<i>n</i>	(global) INTEGER The order of the Hessenberg matrix H (and Z if <i>wantz</i>). $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER It is assumed that the matrix H is already upper triangular in rows and columns $1:ilo-1$ and $ihiz+1:n$. <i>ilo</i> and <i>ihiz</i> are normally set by a previous call to p?gebal , and then passed to p?gehrd when the matrix output by <i>ihiz</i> is reduced to Hessenberg form. Otherwise <i>ilo</i> and <i>ihiz</i> should be set to 1 and n , respectively. If $n > 0$, then $1 \leq ilo \leq ihiz \leq n$. If $n = 0$, then $ilo = 1$ and $ihiz = 0$.
<i>h</i>	REAL for pslaqr0 DOUBLE PRECISION for pdlaqr0 (global) array of size (<i>lld_h</i> , $LOC_c(n)$) The upper Hessenberg matrix H .
<i>desch</i>	(global and local) INTEGER Array of size <i>dlen_</i> .

	The array descriptor for the distributed matrix H .
<i>iloz, ihiz</i>	INTEGER Specify the rows of the matrix Z to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code> , $1 \leq iloz \leq ilo$; $ihz \leq ihiz \leq n$.
<i>z</i>	REAL for <code>pslaqr0</code> DOUBLE PRECISION for <code>pdlaqr0</code> Array of size $(lld_z, LOC_c(n))$. If <i>wantz</i> is <code>.TRUE.</code> , contains the matrix Z . If <i>wantz</i> is <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix Z .
<i>work</i>	REAL for <code>pslaqr0</code> DOUBLE PRECISION for <code>pdlaqr0</code> (local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) INTEGER The length of the workspace array <i>work</i> .
<i>iwork</i>	(local workspace) INTEGER array of size <i>liwork</i>
<i>liwork</i>	(local) INTEGER The length of the workspace array <i>iwork</i> .
<i>recllevel</i>	(local) INTEGER Level of recursion. <i>recllevel</i> = 0 must hold on entry.

OUTPUT Parameters

<i>h</i>	On exit, if <i>wantt</i> is <code>.TRUE.</code> , the matrix H is upper quasi-triangular in rows and columns <i>ilo:ihz</i> , with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $H(i,i) = H(i+1,i+1)$ and $H(i+1,i) \cdot H(i,i+1) < 0$. If <i>info</i> = 0 and <i>wantt</i> is <code>.FALSE.</code> , the contents of <i>h</i> are unspecified on exit.
<i>wr, wi</i>	REAL for <code>pslaqr0</code> DOUBLE PRECISION for <code>pdlaqr0</code> The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihz</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)th, with $wi(i) > 0$ and $wi(i+1) < 0$. If <i>wantt</i> is <code>.TRUE.</code> , the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>h</i> .

<i>z</i>	<p>Updated matrix with transformations applied only to the submatrix $Z(i_{lo}:i_{hi},i_{lo}:i_{hi})$.</p> <p>If <code>COMPZ = 'I'</code>, on exit, if <code>info = 0</code>, <i>z</i> contains the orthogonal matrix <i>Z</i> of the Schur vectors of <i>H</i>.</p> <p>If <code>wantz</code> is <code>.TRUE.</code>, then $Z(i_{lo}:i_{hi},i_{loz}:i_{hiz})$ is replaced by $Z(i_{lo}:i_{hi},i_{loz}:i_{hiz})*U$, when <i>U</i> is the orthogonal/unitary Schur factor of $H(i_{lo}:i_{hi},i_{lo}:i_{hi})$.</p> <p>If <code>wantzis</code> <code>.FALSE.</code>, then <i>z</i> is not defined.</p>
<i>work(1)</i>	On exit, if <code>info = 0</code> , <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <code>info = 0</code> , <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	<p>INTEGER</p> <p>> 0: if <code>info = i</code>, then the routine failed to compute all the eigenvalues. Elements $1:i_{lo}-1$ and $i+1:n$ of <i>wr</i> and <i>wi</i> contain those eigenvalues which have been successfully computed.</p> <p>> 0: if <code>wanttis</code> <code>.FALSE.</code>, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns <i>ilo</i> through <i>ihi</i> of the final output value of <i>h</i>.</p> <p>> 0: if <code>wantt</code> is <code>.TRUE.</code>, then $(\text{initial value of } H)*U = U*(\text{final value of } H)$, where <i>U</i> is an orthogonal/unitary matrix. The final value of <i>H</i> is upper Hessenberg and quasi-triangular/triangular in rows and columns <code>info+1</code> through <i>ihi</i>.</p> <p>> 0: if <code>wantz</code> is <code>.TRUE.</code>, then (final value of $Z(i_{lo}:i_{hi},i_{loz}:i_{hiz})$) = (initial value of $Z(i_{lo}:i_{hi},i_{loz}:i_{hiz})$)*<i>U</i>, where <i>U</i> is the orthogonal/unitary matrix in the previous expression (regardless of the value of <code>wantt</code>).</p> <p>> 0: if <code>wantzis</code> <code>.FALSE.</code>, then <i>z</i> is not accessed.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr1

Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.

Syntax

```
call pslaqr1( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info )
```

```
call pdlaqr1( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info )
```

Description

`p?laqr1` is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

This is a modified version of `p?lahqr` from ScaLAPACK version 1.7.3. The following modifications were made:

- Workspace query functionality was added.
- Aggressive early deflation is implemented.
- Aggressive deflation (looking for two consecutive small subdiagonal elements by PSLACONSB) is abandoned.
- The returned Schur form is now in canonical form, i.e., the returned 2-by-2 blocks really correspond to complex conjugate pairs of eigenvalues.
- For some reason, the original version of `p?lahqr` sometimes did not read out the converged eigenvalues correctly. This is now fixed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>wantt</i>	(global) LOGICAL = .TRUE. : the full Schur form T is required; = .FALSE. : only eigenvalues are required.
<i>wantz</i>	(global) LOGICAL = .TRUE. : the matrix of Schur vectors Z is required; = .FALSE. : Schur vectors are not required.
<i>n</i>	(global) LOGICAL The order of the Hessenberg matrix A (and Z if <i>wantz</i>). $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER It is assumed that the matrix A is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $A(ilo,ilo-1) = 0$ (unless $ilo = 1$). <code>p?laqr1</code> works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of H if <i>wantt</i> is .TRUE.. $1 \leq ilo \leq \max(1, ihi)$; $ihi \leq n$.
<i>a</i>	REAL for <code>pslaqr1</code> DOUBLE PRECISION for <code>pdlqr1</code> (global) array of size $(lld_a, LOC_c(n))$ On entry, the upper Hessenberg matrix A .
<i>desca</i>	(global and local) INTEGER array of size $dlen_$.

	The array descriptor for the distributed matrix <i>A</i> .
<i>iloz, ihiz</i>	(global) INTEGER Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code> . $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n$.
<i>z</i>	REAL for <code>pslaqr1</code> DOUBLE PRECISION for <code>pdlaqr1</code> (global) array of size (<i>lld_z</i> , <i>LOC_c</i> (<i>n</i>)). If <i>wantz</i> is <code>.TRUE.</code> , on entry <i>z</i> must contain the current matrix <i>Z</i> of transformations accumulated by <code>p?hseqr</code> If <i>wantz</i> is <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	REAL for <code>pslaqr1</code> DOUBLE PRECISION for <code>pdlaqr1</code> (local output) array of size <i>lwork</i>
<i>lwork</i>	(local) INTEGER The size of the work array (<i>lwork</i> \geq 1). If <i>lwork</i> =-1, then a workspace query is assumed.
<i>iwork</i>	(global and local) INTEGER array of size <i>ilwork</i> This holds the some of the IBLK integer arrays.
<i>ilwork</i>	(local) INTEGER The size of the <i>iwork</i> array (<i>ilwork</i> \geq 3).

OUTPUT Parameters

<i>a</i>	If <i>wantt</i> is <code>.TRUE.</code> , the matrix <i>A</i> is upper quasi-triangular in rows and columns <i>ilo:ihi</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is <code>.FALSE.</code> , the contents of <i>a</i> are unspecified on exit.
<i>wr, wi</i>	REAL for <code>pslaqr1</code> DOUBLE PRECISION for <code>pdlaqr1</code> (global replicated) array of size <i>n</i> The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)th, with <i>wi</i> (<i>i</i>) > 0

and $w_i(i+1) < 0$. If *wantt* is `.TRUE.`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *a*. *a* may be returned with larger diagonal blocks until the next release.

<i>z</i>	On exit <i>z</i> is updated; transformations are applied only to the submatrix $Z(i_{loz}:i_{hiz}, i_{lo}:i_{hi})$. If <i>wantzis</i> <code>.FALSE.</code> , <i>z</i> is not referenced.
<i>work</i> (1)	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal <i>lwork</i> .
<i>info</i>	(global) INTEGER < 0: parameter number <i>-info</i> incorrect or inconsistent = 0: successful exit > 0: <code>p?laqr1</code> failed to compute all the eigenvalues <i>ilo</i> to <i>ihi</i> in a total of $30*(ihi-ilo+1)$ iterations; if <i>info</i> = <i>i</i> , elements <i>i+1:ihi</i> of <i>wr</i> and <i>wi</i> contain those eigenvalues which have been successfully computed.

Application Notes

This algorithm is very similar to `p?ahqr`. Unlike `p?lahqr`, instead of sending one double shift through the largest unreduced submatrix, this algorithm sends multiple double shifts and spaces them apart so that there can be parallelism across several processor row/columns. Another critical difference is that this algorithm aggregates multiple transforms together in order to apply them in a block fashion.

Current Notes and/or Restrictions:

- This code requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size. Unwise choices of too small a block size can lead to bad performance.
- This code requires *a* and *z* to be distributed identically and have identical contexts.
- This release currently does not have a routine for resolving the Schur blocks into regular 2x2 form after this code is completed. Because of this, a significant performance impact is required while the deflation is done by sometimes a single column of processors.
- This code does not currently block the initial transforms so that none of the rows or columns for any bulge are completed until all are started. To offset pipeline start-up it is recommended that at least $2*LCM(NPROW, NPCOL)$ bulges are used (if possible)
- The maximum number of bulges currently supported is fixed at 32. In future versions this will be limited only by the incoming *work* array.
- The matrix *A* must be in upper Hessenberg form. If elements below the subdiagonal are nonzero, the resulting transforms may be nonsimilar. This is also true with the LAPACK routine.
- For this release, it is assumed `rsrc_ = csrc_ = 0`
- Currently, all the eigenvalues are distributed to all the nodes. Future releases will probably distribute the eigenvalues by the column partitioning.
- The internals of this routine are subject to change.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call pslaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

```
call pdlaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

Description

`p?laqr2` accepts as input an upper Hessenberg matrix A and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output A is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of A . It is to be hoped that the final version of A has many zero subdiagonal entries.

This routine handles small deflation windows which is affordable by one processor. Normally, it is called by `p?laqr1`. All the inputs are assumed to be valid without checking.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>wantt</i>	(global) LOGICAL If <code>.TRUE.</code> , then the Hessenberg matrix A is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine). If <code>.FALSE.</code> , then only enough of A is updated to preserve the eigenvalues.
<i>wantz</i>	(global) LOGICAL If <code>.TRUE.</code> , then the orthogonal matrix Z is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine). If <code>.FALSE.</code> , then z is not referenced.
<i>n</i>	(global) INTEGER The order of the matrix A and (if <i>wantz</i> is <code>.TRUE.</code>) the order of the orthogonal matrix Z .
<i>ktop, kbot</i>	(global) INTEGER It is assumed without a check that either $kbot = n$ or $A(kbot+1, kbot) = 0$. $kbot$ and $ktop$ together determine an isolated block along the diagonal of the Hessenberg matrix. However, $A(ktop, ktop-1) = 0$ is not essentially necessary if <i>wantt</i> is <code>.TRUE.</code> .
<i>nw</i>	(global) INTEGER

	Deflation window size. $1 \leq n_w \leq (k_{bot} - k_{top} + 1)$. Normally $n_w \geq 3$ if <code>p?laqr2</code> is called by <code>p?laqr1</code> .
<code>a</code>	REAL for <code>pslaqr2</code> DOUBLE PRECISION for <code>pdlaqr2</code> (local) array of size $(lld_a, LOC_c(n))$ The initial n -by- n section of <code>a</code> stores the Hessenberg matrix undergoing aggressive early deflation.
<code>desca</code>	(global and local) INTEGER array of size $dlen_.$ The array descriptor for the distributed matrix <code>A</code> .
<code>iloz, ihiz</code>	(global) INTEGER Specify the rows of <code>z</code> to which transformations must be applied if <code>wantz</code> is <code>.TRUE.</code> . $1 \leq iloz \leq ihiz \leq n$.
<code>z</code>	REAL for <code>pslaqr2</code> DOUBLE PRECISION for <code>pdlaqr2</code> Array of size $(lld_z, LOC_c(n))$ If <code>wantz</code> is <code>.TRUE.</code> , then on output, the orthogonal similarity transformation mentioned above has been accumulated into <code>z(iloz:ihiz, kbot:ktop)</code> from the right. If <code>wantz</code> is <code>.FALSE.</code> , then <code>z</code> is unreferenced.
<code>descz</code>	(global and local) INTEGER array of size $dlen_.$ The array descriptor for the distributed matrix <code>Z</code> .
<code>t</code>	REAL for <code>pslaqr2</code> DOUBLE PRECISION for <code>pdlaqr2</code> (local workspace) array of size $ldt * n_w$.
<code>ldt</code>	(local) INTEGER The leading dimension of the array <code>t</code> . $ldt \geq n_w$.
<code>v</code>	REAL for <code>pslaqr2</code> DOUBLE PRECISION for <code>pdlaqr2</code> (local workspace) array of size $ldv * n_w$.
<code>ldv</code>	(local) INTEGER The leading dimension of the array <code>v</code> . $ldv \geq n_w$.
<code>wr, wi</code>	REAL for <code>pslaqr2</code> DOUBLE PRECISION for <code>pdlaqr2</code> (local workspace) array of size k_{bot} .
<code>work</code>	REAL for <code>pslaqr2</code>

DOUBLE PRECISION for pdlaqr2
 (local workspace) array of size *lwork*.

lwork (local) INTEGER
work(*lwork*) is a local array and *lwork* is assumed big enough so that $lwork \geq nw * nw$.

OUTPUT Parameters

a On output *a* has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

z

ns (global) INTEGER
 The number of unconverged (that is, approximate) eigenvalues returned in *sr* and *si* that may be used as shifts by the calling subroutine.

nd (global) INTEGER
 The number of converged eigenvalues uncovered by this subroutine.

sr, si REAL for pslaqr2
 DOUBLE PRECISION for pdlaqr2
 (global) array of size *kbot*
 On output, the real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in *sr*(*kbot-nd-ns+1*) through *sr*(*kbot-nd*) and *si*(*kbot-nd-ns+1*) through *si*(*kbot-nd*), respectively.
 On processor #0, the real and imaginary parts of converged eigenvalues are stored in *sr*(*kbot-nd+1*) through *sr*(*kbot*) and *si*(*kbot-nd+1*) through *si*(*kbot*), respectively. On other processors, these entries are set to zero.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call pslaqr3( wantt, wantz, n, ktop, kbot, nw, h, desch, iloz, ihiz, z, descz, ns, nd, sr, si, v, descv, nh, t, desct, nv, wv, descw, work, lwork, iwork, liwork, recllevel )
call pdlaqr3( wantt, wantz, n, ktop, kbot, nw, h, desch, iloz, ihiz, z, descz, ns, nd, sr, si, v, descv, nh, t, desct, nv, wv, descw, work, lwork, iwork, liwork, recllevel )
```

Description

This subroutine accepts as input an upper Hessenberg matrix H and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

Input Parameters

<i>wantt</i>	(global) LOGICAL	If <code>.TRUE.</code> , then the Hessenberg matrix H is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine). If <code>.FALSE.</code> , then only enough of H is updated to preserve the eigenvalues.
<i>wantz</i>	(global) LOGICAL	If <code>.TRUE.</code> , then the orthogonal matrix Z is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine). If <code>.FALSE.</code> , then z is not referenced.
<i>n</i>	(global) INTEGER	The order of the matrix H and (if <i>wantz</i> is <code>.TRUE.</code>), the order of the orthogonal matrix Z .
<i>ktop</i>	(global) INTEGER	It is assumed that either $ktop = 1$ or $H(ktop,ktop-1)=0$. <i>kbot</i> and <i>ktop</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>kbot</i>	(global) INTEGER	It is assumed without a check that either $kbot = n$ or $H(kbot+1,kbot)=0$. <i>kbot</i> and <i>ktop</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>nw</i>	(global) INTEGER	Deflation window size. $1 \leq nw \leq (kbot-ktop+1)$.
<i>h</i>	REAL for <code>pslaqr3</code> DOUBLE PRECISION for <code>pdlaqr3</code> (local) array of size $(lld_h, LOC_c(n))$	The initial n -by- n section of H stores the Hessenberg matrix undergoing aggressive early deflation.
<i>desch</i>	(global and local) INTEGER array of size <i>dlen_</i> .	The array descriptor for the distributed matrix H .
<i>iloz, ihiz</i>	(global) INTEGER	Specify the rows of the matrix Z to which transformations must be applied if <i>wantz</i> is <code>.TRUE.</code> .. $1 \leq iloz \leq ihiz \leq n$.

<i>z</i>	<p>REAL for <code>pslaqr3</code> DOUBLE PRECISION for <code>pdlaqr3</code> Array of size $(lld_z, LOC_c(n))$ If <i>wantz</i> is <code>.TRUE.</code>, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix $Z(i_{low}:i_{high}, k_{bot}:k_{top})$ from the right. If <i>wantz</i> is <code>.FALSE.</code>, then <i>z</i> is unreferenced.</p>
<i>descz</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>Z</i>.</p>
<i>v</i>	<p>REAL for <code>pslaqr3</code> DOUBLE PRECISION for <code>pdlaqr3</code> (global workspace) array of size $(lld_v, LOC_c(nw))$ An <i>nw</i>-by-<i>nw</i> distributed work array.</p>
<i>descv</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>V</i>.</p>
<i>nh</i>	<p>INTEGER scalar The number of columns of <i>t</i>. $nh \geq nw$.</p>
<i>t</i>	<p>REAL for <code>pslaqr3</code> DOUBLE PRECISION for <code>pdlaqr3</code> (global workspace) array of size $(lld_t, LOC_c(nh))$</p>
<i>desct</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>T</i>.</p>
<i>nv</i>	<p>(global) INTEGER The number of rows of work array <i>wv</i> available for workspace. $nv \geq nw$.</p>
<i>wv</i>	<p>(global workspace) REAL array of size $(lld_w, LOC_c(nw))$</p>
<i>descw</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>wv</i>.</p>
<i>work</i>	<p>(local workspace) REAL array of size <i>lwork</i>.</p>
<i>lwork</i>	<p>(local) INTEGER The size of the work array <i>work</i> ($lwork \geq 1$). $lwork = 2 * nw$ suffices, but greater efficiency may result from larger values of <i>lwork</i>. If $lwork = -1$, then a workspace query is assumed; <code>p?laqr3</code> only estimates the optimal workspace size for the given values of <i>n</i>, <i>nw</i>, <i>k_{top}</i> and <i>k_{bot}</i>. The estimate is returned in <i>work</i>(1). No error message related to <i>lwork</i> is issued by <code>xerbla</code>. Neither <i>h</i> nor <i>z</i> are accessed.</p>

iwork (local workspace) INTEGER array of size *liwork*

liwork (local) INTEGER

The length of the workspace array *iwork* ($liwork \geq 1$).
If $liwork = -1$, then a workspace query is assumed.

OUTPUT Parameters

h On output *h* has been transformed by an orthogonal similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

z IF *wantz* is `.TRUE.`, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix $Z(i_{loz}:i_{hiz}, k_{bot}:k_{top})$ from the right.
If *wantz* is `.FALSE.`, then *z* is unreferenced.

ns (global) INTEGER

The number of unconverged (that is, approximate) eigenvalues returned in *sr* and *si* that may be used as shifts by the calling subroutine.

nd (global) INTEGER

The number of converged eigenvalues uncovered by this subroutine.

sr, si REAL for `pslaqr3`
DOUBLE PRECISION for `pdlaqr3`

(global) array of size *kbot*. The real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in $sr(k_{bot}-nd-ns+1)$ through $sr(k_{bot}-nd)$ and $si(k_{bot}-nd-ns+1)$ through $si(k_{bot}-nd)$, respectively. The real and imaginary parts of converged eigenvalues are stored in $sr(k_{bot}-nd+1)$ through $sr(k_{bot})$ and $si(k_{bot}-nd+1)$ through $si(k_{bot})$, respectively.

work(1) On exit, if *info* = 0, *work*(1) returns the optimal *lwork*

iwork(1) On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.

Syntax

```
call pslaqr4( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, t,
            ldt, v, ldv, work, lwork, info )
call pdlaqr4( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, t,
            ldt, v, ldv, work, lwork, info )
```

Description

`p?laqr4` is an auxiliary routine used to find the Schur decomposition and or eigenvalues of a matrix already in Hessenberg form from cols ilo to ihi . This routine requires that the active block is small enough, i.e. $ihi-ilo+1 \leq ldt$, so that it can be solved by LAPACK. Normally, it is called by `p?laqr1`. All the inputs are assumed to be valid without checking.

Input Parameters

<code>wantt</code>	(global) LOGICAL = <code>.TRUE.</code> : the full Schur form T is required; = <code>.FALSE.</code> : only eigenvalues are required.
<code>wantz</code>	(global) LOGICAL = <code>.TRUE.</code> : the matrix of Schur vectors Z is required; = <code>.FALSE.</code> : Schur vectors are not required.
<code>n</code>	(global) INTEGER The order of the Hessenberg matrix A (and Z if <code>wantz</code>). $n \geq 0$.
<code>ilo, ihi</code>	(global) INTEGER It is assumed that a is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $A(ilo,ilo-1) = 0$ (unless $ilo = 1$). <code>p?laqr4</code> works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of A if <code>wantt</code> is <code>.TRUE.</code> .. $1 \leq ilo \leq \max(1, ihi)$; $ihi \leq n$.
<code>a</code>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (global) array of size $(lld_a, LOC_c(n))$ The upper Hessenberg matrix A .
<code>desca</code>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix a .
<code>iloz, ihiz</code>	(global) INTEGER Specify the rows of the matrix Z to which transformations must be applied if <code>wantz</code> is <code>.TRUE.</code> .. $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.
<code>z</code>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (global) array. If <code>wantz</code> is <code>.TRUE.</code> , on entry z must contain the current matrix Z of transformations accumulated by <code>p?hseqr</code> . If <code>wantz</code> is <code>.FALSE.</code> , z is not referenced.
<code>descz</code>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix Z .

<i>t</i>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (local workspace) array of size $ldt*(ihi-ilo+1)$.
<i>ldt</i>	(local) INTEGER The leading dimension of the array <i>t</i> . $ldt \geq ihi-ilo+1$.
<i>v</i>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (local workspace) array of size $ldv*(ihi-ilo+1)$.
<i>ldv</i>	(local) INTEGER The leading dimension of the array <i>v</i> . $ldv \geq ihi-ilo+1$.
<i>work</i>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (local workspace) array of size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER The size of the work array <i>work</i> . $lwork \geq ihi-ilo+1$.

OUTPUT Parameters

<i>a</i>	On exit, if <i>wantt</i> is <code>.TRUE.</code> , the matrix <i>A</i> is upper quasi-triangular in rows and columns <i>ilo:ihi</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is <code>.FALSE.</code> , the contents of <i>a</i> are unspecified on exit.
<i>wr, wi</i>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (global replicated) array of size <i>n</i> The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)th, with $wi(i) > 0$ and $wi(i+1) < 0$. If <i>wantt</i> is <code>.TRUE.</code> , the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>a</i> . The matrix <i>A</i> may be returned with larger diagonal blocks until the next release.
<i>z</i>	If <i>wantz</i> is <code>.TRUE.</code> , <i>z</i> is updated with transformations applied only to the submatrix $Z(iloz:ihez, ilo:ihi)$.
<i>info</i>	(global) INTEGER < 0: parameter number <i>-info</i> incorrect or inconsistent; = 0: successful exit;

> 0: `p?laqr4` failed to compute all the eigenvalues `ilo` to `ihi` in a total of $30*(ihi-ilo+1)$ iterations; if `info = i`, elements `i+1:ihi` of `wr` and `wi` contain those eigenvalues which have been successfully computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call pslaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz,
ihiz, z, descz, work, lwork, iwork, liwork )
```

```
call pdlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz,
ihiz, z, descz, work, lwork, iwork, liwork )
```

Description

This auxiliary subroutine called by `p?laqr0` performs a single small-bulge multi-shift QR sweep by chasing separated groups of bulges along the main block diagonal of a Hessenberg matrix H .

Input Parameters

<code>wantt</code>	(global) LOGICAL scalar <code>wantt = .TRUE.</code> if the quasi-triangular Schur factor is being computed. <code>wantt</code> is set to <code>.FALSE.</code> otherwise.
<code>wantz</code>	(global) LOGICAL scalar <code>wantz = .TRUE.</code> if the orthogonal Schur factor is being computed. <code>wantz</code> is set to <code>.FALSE.</code> otherwise.
<code>kacc22</code>	(global) INTEGER Value 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: <code>p?laqr5</code> does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: <code>p?laqr5</code> accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: <code>p?laqr5</code> accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<code>n</code>	(global) INTEGER scalar The order of the Hessenberg matrix H and, if <code>wantz</code> is <code>.TRUE.</code> , the order of the orthogonal matrix Z .
<code>ktop, kbot</code>	(global) INTEGER scalar

These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $ktop = 1$ or $H(ktop,ktop-1) = 0$ and either $kbot = n$ or $H(kbot+1,kbot) = 0$.

<i>nshfts</i>	(global) INTEGER scalar <i>nshfts</i> gives the number of simultaneous shifts. <i>nshfts</i> must be positive and even.
<i>sr, si</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (global) Array of size <i>nshfts</i> <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.
<i>h</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) Array of size (<i>lld_h</i> , <i>LOC_c(n)</i>) On input <i>h</i> contains a Hessenberg matrix <i>H</i> .
<i>desch</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>H</i> .
<i>iloz, ihiz</i>	(global) INTEGER Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantzis</i> .TRUE.. $1 \leq iloz \leq ihiz \leq n$
<i>z</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) array of size (<i>lld_z</i> , <i>LOC_c(n)</i>) If <i>wantz</i> = .TRUE., then the QR Sweep orthogonal similarity transformation is accumulated into the matrix <i>Z</i> (<i>iloz:ihiz,kbot:ktop</i>) from the right. If <i>wantz</i> = .FALSE., then <i>z</i> is unreferenced.
<i>descz</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) INTEGER The size of the <i>work</i> array (<i>lwork</i> ≥1). If <i>lwork</i> =-1, then a workspace query is assumed.

iwork (local workspace) INTEGER array of size *liwork*

liwork (local) INTEGER

The size of the *iwork* array ($liwork \geq 1$).

If $liwork = -1$, then a workspace query is assumed.

Output Parameters

h A multi-shift QR sweep with shifts $sr(j) + i * si(j)$ is applied to the isolated diagonal block in rows and columns *ktop* through *kbot* of the matrix *H*.

z If *wantzis* .TRUE., *z* is updated with transformations applied only to the submatrix $Z(ihoz:ihiz, kbot:ktop)$.

work(1) On exit, if *info* = 0, *work*(1) returns the optimal *liwork*.

iwork(1) On exit, if *info* = 0, *iwork*(1) returns the optimal *liwork*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.

Syntax

```
call pslaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pclaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

Description

The *p?laqsy* routine equilibrates a symmetric distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the scaling factors in the vectors *sr* and *sc*. The scaling factors are computed by *p?poequ*.

Input Parameters

uplo (global) CHARACTER. Specifies the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ is to be referenced:

- = 'U': Upper triangular part;
- = 'L': Lower triangular part.

n (global) INTEGER.

The order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.

a (local).

REAL for *pslaqsy*

DOUBLE PRECISION for *pdlaqsy*

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$.

On entry, this array contains the local pieces of the distributed matrix $sub(A)$. On entry, the local pieces of the distributed symmetric matrix $sub(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $sub(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $sub(A)$ is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $sub(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $sub(A)$ is not referenced.

ia, ja

(global) INTEGER.

The row and column indices in the global matrix A indicating the first row and the first column of the matrix $sub(A)$, respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

sr

(local)

REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Array of size $LOCr(m_a)$. The scale factors for $A(ia:ia+m-1, ja:ja+n-1)$. sr is aligned with the distributed matrix A , and replicated across every process column. sr is tied to the distributed matrix A .

sc

(local)

REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Array of size $LOCc(m_a)$. The scale factors for $A(ia:ia+m-1, ja:ja+n-1)$. sc is aligned with the distributed matrix A , and replicated across every process column. sc is tied to the distributed matrix A .

scond

(global). REAL for pslaqsy

DOUBLE PRECISION for pdlaqsy

COMPLEX for pclaqsy

COMPLEX*16 for pzlaqsy.

Ratio of the smallest $sr(i)$ (respectively $sc(j)$) to the largest $sr(i)$ (respectively $sc(j)$), with $ia \leq i \leq ia+n-1$ and $ja \leq j \leq ja+n-1$.

amax (global).
 REAL for *pqlaqsy*
 DOUBLE PRECISION for *pdlaqsy*
 COMPLEX for *pclaqsy*
 COMPLEX*16 for *pzlaqsy*.
 Absolute value of largest distributed submatrix entry.

Output Parameters

a On exit,
 if *equed* = 'Y', the equilibrated matrix:
 $\text{diag}(sr(ia:ia+n-1)) * \text{sub}(A) * \text{diag}(sc(ja:ja+n-1))$.

equed (global) CHARACTER*1.
 Specifies whether or not equilibration was done.
 = 'N': No equilibration.
 = 'Y': Equilibration was done, that is, *sub(A)* has been replaced by:
 $\text{diag}(sr(ia:ia+n-1)) * \text{sub}(A) * \text{diag}(sc(ja:ja+n-1))$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lared1d

Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

```
call pslared1d(n, ia, ja, desc, bycol, byall, work, lwork)
call pdlared1d(n, ia, ja, desc, bycol, byall, work, lwork)
```

Description

The *p?lared1d* routine redistributes a 1D array. It assumes that the input array *bycol* is distributed across rows and that all process column contain the same copy of *bycol*. The output array *byall* is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in *bycol*()

n (global) INTEGER.
 The number of elements to be redistributed. $n \geq 0$.

ia, ja (global) INTEGER. *ia, ja* must be equal to 1.

desc (local) INTEGER array of size 9. A 2D array descriptor, which describes *bycol*.

<i>bycol</i>	(local). REAL for pslared1d DOUBLE PRECISION for pdlared1d COMPLEX for pclared1d COMPLEX*16 for pzlared1d. Distributed block cyclic array of global size n and of local size np . <i>bycol</i> is distributed across the process rows. All process columns are assumed to contain the same value.
<i>work</i>	(local). REAL for pslared1d DOUBLE PRECISION for pdlared1d COMPLEX for pclared1d COMPLEX*16 for pzlared1d. size <i>lwork</i> . Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local) INTEGER. The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}(nb_), 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). REAL for pslared1d DOUBLE PRECISION for pdlared1d COMPLEX for pclared1d COMPLEX*16 for pzlared1d. Global size n , local size n . <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>bycol</i> , but it is replicated across all processes rather than being distributed.
--------------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lared2d

*Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*.*

Syntax

```
call pslared2d(n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d(n, ia, ja, desc, byrow, byall, work, lwork)
```

Description

The p?lared2d routine redistributes a 1D array. It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

<i>n</i>	(global) INTEGER. The number of elements to be redistributed. $n \geq 0$.
<i>ia, ja</i>	(global) INTEGER. <i>ia, ja</i> must be equal to 1.
<i>desc</i>	(local) INTEGER array of size <i>dlen_</i> . A 2D array descriptor, which describes <i>byrow</i> .
<i>byrow</i>	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlarred2d. Distributed block cyclic array of global size <i>n</i> and of local size <i>np</i> . <i>byrow</i> is distributed across the process columns. All process rows are assumed to contain the same value.
<i>work</i>	(local). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlarred2d. size <i>lwork</i> . Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local) INTEGER. The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}(nb_), 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). REAL for pslared2d DOUBLE PRECISION for pdlared2d COMPLEX for pclared2d COMPLEX*16 for pzlarred2d. Global size <i>n</i> , local size <i>n</i> . <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>byrow</i> , but it is replicated across all processes rather than being distributed.
--------------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call pslarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The `p?larf` routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<i>side</i>	(global). CHARACTER. = 'L': form $Q * \text{sub}(C)$, = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$.
<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>v</i>	(local). REAL for <code>pslarf</code> DOUBLE PRECISION for <code>pdlarf</code> COMPLEX for <code>pclarf</code> COMPLEX*16 for <code>pzlarf</code> . Pointer into the local memory to an array of size $(lld_v, *)$, containing the local pieces of the global distributed matrix V representing the Householder transformation Q , $V(iv:iv+m-1, jv)$ if $side = 'L'$ and $incv = 1$, $V(iv, jv:jv+m-1)$ if $side = 'L'$ and $incv = m_v$, $V(iv:iv+n-1, jv)$ if $side = 'R'$ and $incv = 1$, $V(iv, jv:jv+n-1)$ if $side = 'R'$ and $incv = m_v$. The array v is the representation of Q . v is not used if $\tau = 0$.
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global matrix V indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.

<i>descv</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>incv</i>	(global) INTEGER. The global increment for the elements of <i>V</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and <i>m_v</i> . <i>incv</i> must not be zero.
<i>tau</i>	(local). REAL for <i>pslarf</i> DOUBLE PRECISION for <i>pdlarf</i> COMPLEX for <i>pclarf</i> COMPLEX*16 for <i>pzlarf</i> . Array of size <i>LOCc(jv)</i> if <i>incv</i> = 1, and <i>LOCr(iv)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>V</i> .
<i>c</i>	(local). REAL for <i>pslarf</i> DOUBLE PRECISION for <i>pdlarf</i> COMPLEX for <i>pclarf</i> COMPLEX*16 for <i>pzlarf</i> . Pointer into the local memory to an array of size (<i>lld_c</i> , <i>LOCc(jc</i> + <i>n</i> - 1)), containing the local pieces of sub(<i>C</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the matrix sub(<i>C</i>), respectively.
<i>descC</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local). REAL for <i>pslarf</i> DOUBLE PRECISION for <i>pdlarf</i> COMPLEX for <i>pclarf</i> COMPLEX*16 for <i>pzlarf</i> . Array of size <i>lwork</i> . If <i>incv</i> = 1, if <i>side</i> = 'L', if <i>ivcol</i> = <i>iccol</i> , <i>lwork</i> ≥ <i>nqc0</i> else

```

        lwork $\geq$ mpc0 + max( 1, nqc0 )
    end if
else if side = 'R' ,
    lwork $\geq$ nqc0 + max( max( 1, mpc0), numroc(numroc( n+
        icoffc,nb_v,0,0,npcol),nb_v,0,0,lcmq ) )
    end if
else if incv = m_v,
    if side = 'L' ,
        lwork $\geq$ mpc0 + max( max( 1, nqc0 ), numroc(
            numroc(m+iroffc,mb_v,0,0,nprow ),mb_v,0,0, lcmq ) )
    else if side = 'R' ,
        if ivrow = icrow,
            lwork $\geq$ mpc0
        else
            lwork $\geq$ nqc0 + max( 1, mpc0 )
        end if
    end if
end if,

```

where *lcm* is the least common multiple of *nprow* and *npcol* and *lcm* = *ilcm*(*nprow*, *npcol*), *lcmp* = *lcm*/*nprow*, *lcmq* = *lcm*/*npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = *indxg2p*(*ic*, *mb_c*, *myrow*, *rsrc_c*, *nprow*),

iccol = *indxg2p*(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

mpc0 = *numroc*(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *nprow*),

nqc0 = *numroc*(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

c (local).
 On exit, sub(*C*) is overwritten by the $Q \cdot \text{sub}(C)$ if *side* = 'L',
 or sub(*C*) * *Q* if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call pslarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descv, work)
```

```
call pdlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descv, work)
```

```
call pclarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descv, work)
```

```
call pzlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descv, work)
```

Description

The `p?larfb` routine applies a real/complex block reflector Q or its transpose Q^T /conjugate transpose Q^H to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left; if <i>side</i> = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Right.
<i>trans</i>	(global) CHARACTER. if <i>trans</i> = 'N': no transpose, apply Q ; for real flavors, if <i>trans</i> ='T': transpose, apply Q^T for complex flavors, if <i>trans</i> = 'C': conjugate transpose, apply Q^H ;
<i>direct</i>	(global) CHARACTER. Indicates how Q is formed from a product of elementary reflectors. if <i>direct</i> = 'F': $Q = H(1)*H(2)*\dots*H(k)$ (Forward) if <i>direct</i> = 'B': $Q = H(k)*\dots*H(2)*H(1)$ (Backward)
<i>storev</i>	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if <i>storev</i> = 'C': Columnwise if <i>storev</i> = 'R': Rowwise.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<i>k</i>	(global) INTEGER. The order of the matrix T .
<i>v</i>	(local).

REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of size

(lld_v , $LOCc(jv+k-1)$) if $storev = 'C'$,
 (lld_v , $LOCc(jv+m-1)$) if $storev = 'R'$ and $side = 'L'$,
 (lld_v , $LOCc(jv+n-1)$) if $storev = 'R'$ and $side = 'R'$.

It contains the local pieces of the distributed vectors V representing the Householder transformation.

if $storev = 'C'$ and $side = 'L'$, $lld_v \geq \max(1, LOCr(iv+m-1))$;

if $storev = 'C'$ and $side = 'R'$, $lld_v \geq \max(1, LOCr(iv+n-1))$;

if $storev = 'R'$, $lld_v \geq LOCr(jv+k-1)$.

iv, jv

(global) INTEGER.

The row and column indices in the global matrix V indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.

descv

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V .

c

(local).

REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.

Pointer into the local memory to an array of size (lld_c , $LOCc(jc+n-1)$), containing the local pieces of $\text{sub}(C)$.

ic, jc

(global) INTEGER. The row and column indices in the global matrix C indicating the first row and the first column of the matrix $\text{sub}(C)$, respectively.

descC

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local).

REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.

Workspace array of size $lwork$.

If $storev = 'C'$,

```

if side = 'L',
  lwork ≥ ( nqc0 + mpc0 ) * k
else if side = 'R',
  lwork ≥ ( nqc0 + max( npv0 + numroc( numroc( n +
    icoffc, nb_v, 0, 0, npc0 ), nb_v, 0, 0, lcmq ),
    mpc0 ) ) * k
end if
else if storev = 'R' ,
  if side = 'L' ,
    lwork ≥ ( mpc0 + max( mqv0 + numroc( numroc( m +
      iroffc, mb_v, 0, 0, nprow ), mb_v, 0, 0, lcmq ),
      nqc0 ) ) * k
  else if side = 'R',
    lwork ≥ ( mpc0 + nqc0 ) * k
  end if
end if,

```

where

```

lcmq = lcm / npc0 with lcm = ilcm( nprow, npc0 ),
iroffv = mod( iv-1, mb_v ), icoffv = mod( jv-1, nb_v ),
ivrow = indxg2p( iv, mb_v, myrow, rsrc_v, nprow ),
ivcol = indxg2p( jv, nb_v, mycol, csrc_v, npc0 ),
MqV0 = numroc( m+icoffv, nb_v, mycol, ivcol, npc0 ),
NpV0 = numroc( n+iroffv, mb_v, myrow, ivrow, nprow ),
iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npc0 ),
Mpc0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),
Npc0 = numroc( n+icoffc, mb_c, myrow, icrow, nprow ),
Nqc0 = numroc( n+icoffc, nb_c, mycol, iccol, npc0 ),

```

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

t

(local).

REAL for *pslarfb*

DOUBLE PRECISION for *pdlarfb*

COMPLEX for *pclarfb*

COMPLEX*16 for pzlarfb.

Array of size (mb_v, mb_v) if $storev = 'R'$, and (nb_v, nb_v) if $storev = 'C'$. The triangular matrix t is the representation of the block reflector.

c

(local).

On exit, $sub(C)$ is overwritten by the $Q*sub(C)$, or $Q'*sub(C)$, or $sub(C)*Q$, or $sub(C)*Q'$. Q' is transpose (conjugate transpose) of Q .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

```
call pclarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

```
call pzlarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The `p?larfc` routine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $sub(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau * v * v',$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

$side$

(global) CHARACTER.

if $side = 'L'$: form $Q^H * sub(C)$;

if $side = 'R'$: form $sub(C) * Q^H$.

m

(global) INTEGER.

The number of rows in the distributed matrix $sub(C)$. ($m \geq 0$).

n

(global) INTEGER.

The number of columns in the distributed matrix $sub(C)$. ($n \geq 0$).

v

(local).

COMPLEX for pclarfc

COMPLEX*16 for pzlarfc.

Pointer into the local memory to an array of size $(ld_v, *)$, containing the local pieces of the global distributed matrix V representing the Householder transformation Q ,

$V(iv:iv+m-1, jv)$ if $side = 'L'$ and $incv = 1$,

$V(iv, jv:jv+m-1)$ if $side = 'L'$ and $incv = m_v$,

$V(iv:iv+n-1, jv)$ if $side = 'R'$ and $incv = 1$,

$V(iv, jv:jv+n-1)$ if $side = 'R'$ and $incv = m_v$.

The array v is the representation of Q . v is not used if $tau = 0$.

iv, jv

(global) INTEGER.

The row and column indices in the global matrix V indicating the first row and the first column of the matrix $sub(V)$, respectively.

$descv$

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V .

$incv$

(global) INTEGER.

The global increment for the elements of v . Only two values of $incv$ are supported in this version, namely 1 and m_v .

$incv$ must not be zero.

tau

(local)

COMPLEX for `pclarfc`

COMPLEX*16 for `pzlarfc`.

Array of size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix V .

c

(local).

COMPLEX for `pclarfc`

COMPLEX*16 for `pzlarfc`.

Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$, containing the local pieces of $sub(C)$.

ic, jc

(global) INTEGER.

The row and column indices in the global matrix C indicating the first row and the first column of the matrix $sub(C)$, respectively.

$descC$

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

$work$

(local).

COMPLEX for `pclarfc`

COMPLEX*16 for `pzlarfc`.

Workspace array of size $lwork$.

If $incv = 1$,

if $side = 'L'$,

if $ivcol = iccol$,

$lwork \geq nqc0$

```

else
   $lwork \geq mpc0 + \max(1, nqc0)$ 
end if
else if  $side = 'R'$ ,
   $lwork \geq nqc0 + \max(\max(1, mpc0), \text{numroc}(\text{numroc}(n+icoffc, nb\_v, 0, 0, npc0), nb\_v, 0, 0, lcmq))$ 
end if
else if  $incv = m\_v$ ,
  if  $side = 'L'$ ,
     $lwork \geq mpc0 + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m+iroffc, mb\_v, 0, 0, nprow), mb\_v, 0, 0, lcmp))$ 
  else if  $side = 'R'$ ,
    if  $ivrow = icrow$ ,
       $lwork \geq mpc0$ 
    else
       $lwork \geq nqc0 + \max(1, mpc0)$ 
    end if
  end if
end if,

```

where lcm is the least common multiple of $nprow$ and $npcol$ and $lcm = \text{ilcm}(nprow, npc0)$,

$lcmp = lcm/nprow, lcmq = lcm/npcol$,

$iroffc = \text{mod}(ic-1, mb_c), icoffc = \text{mod}(jc-1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,

$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0)$,

$mpc0 = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprow)$,

$nqc0 = \text{numroc}(n+icoffc, nb_c, mycol, iccol, npc0)$,

$\text{ilcm}, \text{indxg2p}$, and numroc are ScaLAPACK tool functions; $myrow, mycol, nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

c

(local).

On exit, $\text{sub}(C)$ is overwritten by the $Q^H * \text{sub}(C)$ if $side = 'L'$, or $\text{sub}(C) * Q^H$ if $side = 'R'$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call pslarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pdlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pclarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pzlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
```

Description

The p?larfg routine generates a real/complex elementary reflector H of order n , such that

$$H * \text{sub}(X) = H * \begin{pmatrix} x(iax, jax) \\ x \end{pmatrix} = \begin{pmatrix} alpha \\ 0 \end{pmatrix}, H^* H = I,$$

where $alpha$ is a scalar (a real scalar - for complex flavors), and $\text{sub}(X)$ is an $(n-1)$ -element real/complex distributed vector $X(ix:ix+n-2, jx)$ if $incx = 1$ and $X(ix, jx:jx+n-2)$ if $incx = m_x$. H is represented in the form

$$H = I - tau * \begin{pmatrix} 1 \\ v \end{pmatrix} * (1 v')$$

where tau is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that H is not Hermitian.

If the elements of $\text{sub}(X)$ are all zero (and $X(iax, jax)$ is real for complex flavors), then $tau = 0$ and H is taken to be the unit matrix.

Otherwise $1 \leq \text{real}(tau) \leq 2$ and $\text{abs}(tau-1) \leq 1$.

Input Parameters

n	(global) INTEGER. The global order of the elementary reflector. $n \geq 0$.
iax, jax	(global) INTEGER. The global row and column indices of $X(iax, jax)$ in the global matrix X .
x	(local). REAL for pslarfg DOUBLE PRECISION for pdlarfg COMPLEX for pclarfg COMPLEX*16 for pzlarfg.

Pointer into the local memory to an array of size $(lld_x, *)$. This array contains the local pieces of the distributed vector $sub(X)$. Before entry, the incremented array $sub(X)$ must contain vector x .

ix, jx (global) INTEGER.

The row and column indices in the global matrix X indicating the first row and the first column of $sub(X)$, respectively.

descx (global and local) INTEGER.

Array of size $dlen_$. The array descriptor for the distributed matrix X .

incx (global) INTEGER.

The global increment for the elements of x . Only two values of *incx* are supported in this version, namely 1 and m_x . *incx* must not be zero.

Output Parameters

alpha (local)

REAL for pslafg

DOUBLE PRECISION for pdlafg

COMPLEX for pclafg

COMPLEX*16 for pzlafg.

On exit, *alpha* is computed in the process scope having the vector $sub(X)$.

x (local).

On exit, it is overwritten with the vector v .

tau (local).

REAL for pslarfg

DOUBLE PRECISION for pdlarfg

COMPLEX for pclarfg

COMPLEX*16 for pzlarfg.

Array of size $LOCc(jx)$ if $incx = 1$, and $LOCr(ix)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix X .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larft

Forms the triangular vector T of a block reflector $H=I-V^*T^*V^H$.

Syntax

```
call pslarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

```
call pdlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

call pclarft(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pzlarft(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

Description

The p?larft routine forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If *direct* = 'F', $H = H(1)*H(2)*\dots*H(k)$, and T is upper triangular;

If *direct* = 'B', $H = H(k)*\dots*H(2)*H(1)$, and T is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the distributed matrix V , and

$$H = I - V * T * V'$$

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the distributed matrix V , and

$$H = I - V' * T * V.$$

Input Parameters

<i>direct</i>	(global) CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if <i>direct</i> = 'F': $H = H(1)*H(2)*\dots*H(k)$ (forward) if <i>direct</i> = 'B': $H = H(k)*\dots*H(2)*H(1)$ (backward).
<i>storev</i>	(global) CHARACTER*1. Specifies how the vectors that define the elementary reflectors are stored (See <i>Application Notes</i> below): if <i>storev</i> = 'C': columnwise; if <i>storev</i> = 'R': rowwise.
<i>n</i>	(global) INTEGER. The order of the block reflector H . $n \geq 0$.
<i>k</i>	(global) INTEGER. The order of the triangular factor T , is equal to the number of elementary reflectors. $1 \leq k \leq mb_v$ ($= nb_v$).
<i>v</i>	REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Pointer into the local memory to an array of local size $(LOCr(iv+n-1), LOCc(jv+k-1))$ if <i>storev</i> = 'C', and $(LOCr(iv+k-1), LOCc(jv+n-1))$ if <i>storev</i> = 'R'.

The distributed matrix V contains the Householder vectors. (See *Application Notes* below).

<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global matrix V indicating the first row and the first column of the matrix sub(V), respectively.
<i>descv</i>	(local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix V .
<i>tau</i>	(local) REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Array of size $LOCr(iv+k-1)$ if $incv = m_v$, and $LOCc(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix V .
<i>work</i>	(local). REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Workspace array of size $k*(k-1)/2$.

Output Parameters

<i>v</i>	REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft.
<i>t</i>	(local) REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Array of size (nb_v, nb_v) if $storev = 'C'$, and (mb_v, mb_v) otherwise. It contains the k -by- k triangular factor of the block reflector associated with v . If $direct = 'F'$, t is upper triangular; if $direct = 'B'$, t is lower triangular.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': form $Q * \text{sub}(C)$, if <i>side</i> = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$ (for real flavors).
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<i>l</i>	(global) INTEGER. The columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$, if <i>side</i> = 'R', $n \geq l \geq 0$.
<i>v</i>	(local). REAL for pslarz DOUBLE PRECISION for pdlarz COMPLEX for pclarz COMPLEX*16 for pzlarz. Pointer into the local memory to an array of size (<i>lld_v</i> ,*) containing the local pieces of the global distributed matrix <i>V</i> representing the Householder transformation <i>Q</i> , $V(i_v:i_v+l-1, j_v)$ if <i>side</i> = 'L' and <i>incv</i> = 1, $V(i_v, j_v:j_v+l-1)$ if <i>side</i> = 'L' and <i>incv</i> = <i>m_v</i> , $V(i_v:i_v+l-1, j_v)$ if <i>side</i> = 'R' and <i>incv</i> = 1, $V(i_v, j_v:j_v+l-1)$ if <i>side</i> = 'R' and <i>incv</i> = <i>m_v</i> . The vector <i>v</i> in the representation of <i>Q</i> . <i>v</i> is not used if <i>tau</i> = 0.
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global distributed matrix <i>V</i> indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.
<i>descv</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>incv</i>	(global) INTEGER. The global increment for the elements of <i>V</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and <i>m_v</i> . <i>incv</i> must not be zero.
<i>tau</i>	(local) REAL for pslarz DOUBLE PRECISION for pdlarz

COMPLEX for pclarz
 COMPLEX*16 for pzlarz.
 Array of size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.
tau is tied to the distributed matrix *V*.

c (local).
 REAL for pslarz
 DOUBLE PRECISION for pdlarz
 COMPLEX for pclarz
 COMPLEX*16 for pzlarz.
 Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$, containing the local pieces of sub(*C*).

ic, jc (global) INTEGER.
 The row and column indices in the global matrix *C* indicating the first row and the first column of the matrix sub(*C*), respectively.

desc (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work (local).
 REAL for pslarz
 DOUBLE PRECISION for pdlarz
 COMPLEX for pclarz
 COMPLEX*16 for pzlarz.
 Array of size *lwork*
 If $incv = 1$,
 if $side = 'L'$,
 if $ivcol = iccol$,
 $lwork \geq NqC0$
 else
 $lwork \geq MpC0 + \max(1, NqC0)$
 end if
 else if $side = 'R'$,
 $lwork \geq NqC0 + \max(\max(1, MpC0), \text{numroc}(\text{numroc}(n+icoffc, nb_v, 0, 0, npcol), nb_v, 0, 0, lcmq))$
 end if
 else if $incv = m_v$,
 if $side = 'L'$,
 $lwork \geq MpC0 + \max(\max(1, NqC0), \text{numroc}(\text{numroc}(m+iroffc, mb_v, 0, 0, nprow), mb_v, 0, 0, icmp))$

```

else if side = 'R' ,
  if ivrow = icrow,
    lwork ≥ MpC0
  else
    lwork ≥ NqC0 + max(1, MpC0)
  end if
end if
end if.

```

Here *lcm* is the least common multiple of *npro*w and *npcol* and

lcm = *ilcm*(*npro*w, *npcol*), *lcmp* = *lcm* / *npro*w,

lcmq = *lcm* / *npcol*,

iroffc = mod(*ic*-1, *mb_c*), *icoffc* = mod(*jc*-1, *nb_c*),

icrow = *indxg2p*(*ic*, *mb_c*, *myrow*, *rsrc_c*, *npro*w),

iccol = *indxg2p*(*jc*, *nb_c*, *mycol*, *csrc_c*, *npcol*),

mpc0 = *numroc*(*m*+*iroffc*, *mb_c*, *myrow*, *icrow*, *npro*w),

nqc0 = *numroc*(*n*+*icoffc*, *nb_c*, *mycol*, *iccol*, *npcol*),

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *npro*w, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

c

(local).

On exit, *sub*(*C*) is overwritten by the *Q***sub*(*C*) if *side* = 'L', or *sub*(*C*)**Q* if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by *p?tzrzf* to a general matrix.

Syntax

```
call pslarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

```
call pdlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

```
call pclarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

```
call pzlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

Description

The `p?larzbroutine` applies a real/complex block reflector Q or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Q is a product of k elementary reflectors as returned by `p?tzzrf`.

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<code>side</code>	(global) CHARACTER. if <code>side = 'L'</code> : apply Q or Q^T (Q^H for complex flavors) from the Left; if <code>side = 'R'</code> : apply Q or Q^T (Q^H for complex flavors) from the Right.
<code>trans</code>	(global) CHARACTER. if <code>trans = 'N'</code> : No transpose, apply Q ; If <code>trans='T'</code> : Transpose, apply Q^T (real flavors); If <code>trans='C'</code> : Conjugate transpose, apply Q^H (complex flavors).
<code>direct</code>	(global) CHARACTER. Indicates how H is formed from a product of elementary reflectors. if <code>direct = 'F'</code> : $H = H(1)*H(2)*\dots*H(k)$ - forward (not supported) ; if <code>direct = 'B'</code> : $H = H(k)*\dots*H(2)*H(1)$ - backward.
<code>storev</code>	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if <code>storev = 'C'</code> : columnwise (not supported) . if <code>storev = 'R'</code> : rowwise.
<code>m</code>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<code>k</code>	(global) INTEGER. The order of the matrix T . (= the number of elementary reflectors whose product defines the block reflector).
<code>l</code>	(global) INTEGER. The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <code>side = 'L'</code> , $m \geq l \geq 0$, if <code>side = 'R'</code> , $n \geq l \geq 0$.
<code>v</code>	(local).

REAL for pslarzb
 DOUBLE PRECISION for pdlarzb
 COMPLEX for pclarzb
 COMPLEX*16 for pzlarzb.

Pointer into the local memory to an array of size $(lld_v, LOCc(jv+m-1))$ if $side = 'L'$, $(lld_v, LOCc(jv+m-1))$ if $side = 'R'$.

It contains the local pieces of the distributed vectors V representing the Householder transformation as returned by `p?tzrzf`.

$lld_v \geq LOCr(iv+k-1)$.

iv, jv (global) INTEGER.
 The row and column indices in the global matrix V indicating the first row and the first column of the submatrix $sub(V)$, respectively.

descv (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V .

t (local)
 REAL for pslarzb
 DOUBLE PRECISION for pdlarzb
 COMPLEX for pclarzb
 COMPLEX*16 for pzlarzb.
 Array of size mb_v by mb_v .
 The lower triangular matrix T in the representation of the block reflector.

c (local).
 REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.
 Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$.
 On entry, the m -by- n distributed matrix $sub(C)$.

ic, jc (global) INTEGER.
 The row and column indices in the global matrix C indicating the first row and the first column of the submatrix $sub(C)$, respectively.

desc (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work (local).
 REAL for pslarzb
 DOUBLE PRECISION for pdlarzb
 COMPLEX for pclarzb

COMPLEX*16 for pzlarzb.

Array of size *lwork*.

```

If storev = 'C' ,
    if side = 'L' ,
        lwork ≥ (nqc0 + mpc0) * k
    else if side = 'R' ,
        lwork ≥ (nqc0 + max(npv0 + numroc(numroc(n+icoffc, nb_v, 0, 0,
npcol),
        nb_v, 0, 0, lcmq), mpc0)) * k
    end if
else if storev = 'R' ,
    if side = 'L' ,
        lwork ≥ (mpc0 + max(mqv0 + numroc(numroc(m+iroffc, mb_v, 0, 0, nprow),
        mb_v, 0, 0, lcmp), nqc0)) * k
    else if side = 'R' ,
        lwork ≥ (mpc0 + nqc0) * k
    end if
end if.

```

Here $lcmq = lcm/npcol$ with $lcm = iclm(nprow, npcol)$,
 $iroffv = \text{mod}(iv-1, mb_v)$, $icoffv = \text{mod}(jv-1, nb_v)$,
 $ivrow = \text{indxg2p}(iv, mb_v, myrow, rsrc_v, nprow)$,
 $ivcol = \text{indxg2p}(jv, nb_v, mycol, csrc_v, npcol)$,
 $mqv0 = \text{numroc}(m+icoffv, nb_v, mycol, ivcol, npcol)$,
 $npv0 = \text{numroc}(n+iroffv, mb_v, myrow, ivrow, nprow)$,
 $iroffc = \text{mod}(ic-1, mb_c)$, $icoffc = \text{mod}(jc-1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,
 $iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npcol)$,
 $mpc0 = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprow)$,
 $npc0 = \text{numroc}(n+icoffc, mb_c, myrow, icrow, nprow)$,
 $nqc0 = \text{numroc}(n+icoffc, nb_c, mycol, iccol, npcol)$,
 $ilcm, \text{indxg2p}$, and numroc are ScaLAPACK tool functions; $myrow, mycol, nprow,$ and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

c

(local).

On exit, $\text{sub}(C)$ is overwritten by the $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q'$, where Q' is the transpose (conjugate transpose) of Q .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.

Syntax

```
call pclarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

Description

The p?larzcroutine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by p?tzrzf.

Input Parameters

<i>side</i>	(global) CHARACTER. if <i>side</i> = 'L': form $Q^H * \text{sub}(C)$; if <i>side</i> = 'R': form $\text{sub}(C) * Q^H$.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<i>l</i>	(global) INTEGER. The columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$, if <i>side</i> = 'R', $n \geq l \geq 0$.
<i>v</i>	(local). COMPLEX for pclarzc COMPLEX*16 for pzlarzc.

Pointer into the local memory to an array of size $(lld_v,*)$ containing the local pieces of the global distributed matrix V representing the Householder transformation Q ,

$V(iv:iv+1-1, jv)$ if $side = 'L'$ and $incv = 1$,

$V(iv, jv:jv+1-1)$ if $side = 'L'$ and $incv = m_v$,

$V(iv:iv+1-1, jv)$ if $side = 'R'$ and $incv = 1$,

$V(iv, jv:jv+1-1)$ if $side = 'R'$ and $incv = m_v$.

The vector v in the representation of Q . v is not used if $tau = 0$.

iv, jv

(global) INTEGER.

The row and column indices in the global matrix V indicating the first row and the first column of the matrix $sub(V)$, respectively.

descv

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix V .

incv

(global) INTEGER.

The global increment for the elements of V . Only two values of $incv$ are supported in this version, namely 1 and m_v .

incv must not be zero.

tau

(local)

COMPLEX for `pclarzc`

COMPLEX*16 for `pzlarzc`.

Array of size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

tau is tied to the distributed matrix V .

c

(local).

COMPLEX for `pclarzc`

COMPLEX*16 for `pzlarzc`.

Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$, containing the local pieces of $sub(C)$.

ic, jc

(global) INTEGER.

The row and column indices in the global matrix C indicating the first row and the first column of the matrix $sub(C)$, respectively.

descC

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local).

```
If incv = 1,
  if side = 'L' ,
    if ivcol = iccol,
      lwork ≥ nqc0
    else
```

```

        lwork ≥ mpc0 + max(1, nqc0)
    end if
    else if side = 'R' ,
        lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n+icoffc, nb_v, 0, 0,
        npc0),
            nb_v, 0, 0, lcmq)) end if
    else if incv = m_v,
        if side = 'L' ,
            lwork ≥ mpc0 + max(max(1, nqc0), numroc(numroc(m+iroffc, mb_v, 0, 0,
            nprow),
                mb_v, 0, 0, lcmp))
        else if side = 'R',
            if ivrow = icrow,
                lwork ≥ mpc0
            else
                lwork ≥ nqc0 + max(1, mpc0)
            end if
        end if
    end if
end if
end if

```

Here *lcm* is the least common multiple of *nprow* and *npcol*;

$lcm = ilcm(nprow, npc0), lcmp = lcm/nprow, lcmq = lcm/npcol,$

$iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1, nb_c),$

$icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),$

$iccol = indxg2p(jc, nb_c, mycol, csrc_c, npc0),$

$mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),$

$nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npc0),$

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions;

myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

c

(local).

On exit, sub(*C*) is overwritten by the $Q^H \cdot \text{sub}(C)$ if *side* = 'L', or $\text{sub}(C) \cdot Q^H$ if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzt

Forms the triangular factor *T* of a block reflector $H = I - V \cdot T \cdot V^H$ as returned by p?ttrzf.

Syntax

call pslarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pdlarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pclarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

call pzlarzt(*direct*, *storev*, *n*, *k*, *v*, *iv*, *jv*, *descv*, *tau*, *t*, *work*)

Description

The `p?larztroutine` forms the triangular factor T of a real/complex block reflector H of order greater than n , which is defined as a product of k elementary reflectors as returned by `p?tzrzf`.

If `direct = 'F'`, $H = H(1)*H(2)*\dots*H(k)$, and T is upper triangular;

If `direct = 'B'`, $H = H(k)*\dots*H(2)*H(1)$, and T is lower triangular.

If `storev = 'C'`, the vector which defines the elementary reflector $H(i)$, is stored in the i -th column of the array v , and

$$H = i-v*t*v'$$

If `storev = 'R'`, the vector, which defines the elementary reflector $H(i)$, is stored in the i -th row of the array v , and

$$H = i-v'*t*v$$

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<code>direct</code>	(global) CHARACTER. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if <code>direct = 'F'</code> : $H = H(1)*H(2)*\dots*H(k)$ (Forward, not supported) if <code>direct = 'B'</code> : $H = H(k)*\dots*H(2)*H(1)$ (Backward).
<code>storev</code>	(global) CHARACTER. Specifies how the vectors which defines the elementary reflectors are stored: if <code>storev = 'C'</code> : columnwise (not supported); if <code>storev = 'R'</code> : rowwise.
<code>n</code>	(global) INTEGER. The order of the block reflector H . $n \geq 0$.
<code>k</code>	(global) INTEGER. The order of the triangular factor T (= the number of elementary reflectors). $1 \leq k \leq mb_v (= nb_v)$.
<code>v</code>	REAL for <code>pslarzt</code> DOUBLE PRECISION for <code>pdlarzt</code> COMPLEX for <code>pclarzt</code> COMPLEX*16 for <code>pzlarzt</code> . Pointer into the local memory to an array of local size $(LOCr(iv+k-1), LOCc(jv+n-1))$. The distributed matrix V contains the Householder vectors. See <i>Application Notes</i> below.

<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global matrix V indicating the first row and the first column of the matrix sub(V), respectively.
<i>descv</i>	(local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix V .
<i>tau</i>	(local) REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Array of size $LOCr(iv+k-1)$ if $incv = m_v$, and $LOCc(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix V .
<i>work</i>	(local). REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Workspace array of size $(k*(k-1)/2)$.

Output Parameters

<i>v</i>	REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt.
<i>t</i>	(local) REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzlarzt. Array of size mb_v by mb_v . It contains the k -by- k triangular factor of the block reflector associated with v . t is lower triangular.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct='F' and storev='C'

$$\begin{matrix}
 \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix} \\
 v = \begin{matrix} . & . & . \\ . & . & . \\ 1 & . & . \\ & 1 & . \\ & & 1 \end{matrix}
 \end{matrix}$$

direct='F' and storev='R':

$$\begin{bmatrix} \overbrace{v1 \ v1 \ v1 \ v1 \ v1}^{\times} \ . \ . \ . \ . \ . \ 1 \\ v2 \ v2 \ v2 \ v2 \ v2 \ . \ . \ . \ 1 \\ v3 \ v3 \ v3 \ v3 \ v3 \ . \ . \ 1 \end{bmatrix}$$

direct='B' and storev='C':

$$\begin{matrix}
 1 \\
 . \ 1 \\
 . \ . \ 1 \\
 . \ . \ . \\
 v = \ . \ . \ . \\
 \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}
 \end{matrix}$$

direct='B' and storev='R':

$$\begin{bmatrix} 1 \ . \ . \ . \ . \ \overbrace{v1 \ v1 \ v1 \ v1 \ v1}^{\times} \\ . \ 1 \ . \ . \ . \ v2 \ v2 \ v2 \ v2 \ v2 \\ . \ . \ 1 \ . \ . \ v3 \ v3 \ v3 \ v3 \ v3 \end{bmatrix}$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .

Syntax

```

call pslascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)

```

call pzlascl(*type*, *cfrom*, *cto*, *m*, *n*, *a*, *ia*, *ja*, *desca*, *info*)

Description

The pzlascl routine multiplies the m -by- n real/complex distributed matrix sub(A) denoting $A(ia:ia+m-1, ja:ja+n-1)$ by the real/complex scalar $cto/cfrom$. This is done without over/underflow as long as the final result $cto*A(i,j)/cfrom$ does not over/underflow. *type* specifies that sub(A) may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

<i>type</i>	(global) CHARACTER. <i>type</i> indicates the storage type of the input distributed matrix. if <i>type</i> = 'G': sub(A) is a full matrix, if <i>type</i> = 'L': sub(A) is a lower triangular matrix, if <i>type</i> = 'U': sub(A) is an upper triangular matrix, if <i>type</i> = 'H': sub(A) is an upper Hessenberg matrix.
<i>cfrom</i> , <i>cto</i>	(global) REAL for pslascl/pclascl DOUBLE PRECISION for pdlascl/pzlascl. The distributed matrix sub(A) is multiplied by $cto/cfrom$. $A(i,j)$ is computed without over/underflow if the final result $cto*A(i,j)/cfrom$ can be represented without over/underflow. <i>cfrom</i> must be nonzero.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(A). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(A). ($n \geq 0$).
<i>a</i>	(local input/local output) REAL for pslascl DOUBLE PRECISION for pdlascl COMPLEX for pclascl COMPLEX*16 for pzlascl. Pointer into the local memory to an array of size ($lld_a, LOCC(ja+n-1)$). This array contains the local pieces of the distributed matrix sub(A).
<i>ia</i> , <i>ja</i>	(global) INTEGER. The column and row indices in the global matrix A indicating the first row and column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) INTEGER. Array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

<i>a</i>	(local). On exit, this array contains the local pieces of the distributed matrix multiplied by <i>cto/cfrom</i> .
<i>info</i>	(local) INTEGER. if <i>info</i> = 0: the execution is successful. if <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laset

Initializes the offdiagonal elements of a matrix to alpha and the diagonal elements to beta.

Syntax

```
call pslaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
```

Description

The *p?laset* routine initializes an *m*-by-*n* distributed matrix sub(*A*) denoting $A(ia:ia+m-1, ja:ja+n-1)$ to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the part of the distributed matrix sub(<i>A</i>) to be set: if <i>uplo</i> = 'U': upper triangular part; the strictly lower triangular part of sub(<i>A</i>) is not changed; if <i>uplo</i> = 'L': lower triangular part; the strictly upper triangular part of sub(<i>A</i>) is not changed. Otherwise: all of the matrix sub(<i>A</i>) is set.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(<i>A</i>). ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>A</i>). ($n \geq 0$).
<i>alpha</i>	(global).

REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzaset.

The constant to which the offdiagonal elements are to be set.

beta

(global).

REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzaset.

The constant to which the diagonal elements are to be set.

Output Parameters

a

(local).

REAL for pslaset
 DOUBLE PRECISION for pdlaset
 COMPLEX for pclaset
 COMPLEX*16 for pzaset.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$.

This array contains the local pieces of the distributed matrix sub(*A*) to be set. On exit, the leading *m*-by-*n* matrix sub(*A*) is set as follows:

if *uplo* = 'U', $A(ia+i-1, ja+j-1) = alpha, 1 \leq j-1, 1 \leq j \leq n,$

if *uplo* = 'L', $A(ia+i-1, ja+j-1) = alpha, j+1 \leq i \leq m, 1 \leq j \leq n,$

otherwise, $A(ia+i-1, ja+j-1) = alpha, 1 \leq i \leq m, 1 \leq j \leq n, ia+i \neq ja+j,$ and, for all *uplo*, $A(ia+i-1, ja+i-1) = beta, 1 \leq i \leq \min(m,n).$

ia, ja

(global) INTEGER.

The column and row indices in the distributed matrix *A* indicating the first row and column of the matrix sub(*A*), respectively.

desca

(global and local) INTEGER.

Array of size *dlen*_. The array descriptor for the distributed matrix *A*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

call pslasmsub(*a, desca, i, l, k, smlnum, buf, lwork*)

call `pdlasmsub(a, desca, i, l, k, smlnum, buf, lwork)`

Description

The `p?lasmsub` routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine does a global maximum and must be called by all processes.

Input Parameters

<i>a</i>	(global) REAL for <code>pslasmsub</code> DOUBLE PRECISION for <code>pdlasmsub</code> Array of size $(lld_a, *)$. On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of size $dlen_$. The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER. The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>smlnum</i>	(global) REAL for <code>pslasmsub</code> DOUBLE PRECISION for <code>pdlasmsub</code> On entry, a "small number" for the given matrix. Unchanged on exit. A suggested value for <i>smlnum</i> is <code>slamch('s') * (n/slamch('p'))</code> for <code>pslasmsub</code> or <code>dlamch('s') * (n/dlamch('p'))</code> for <code>pdlasmsub</code> . See lamch .
<i>lwork</i>	(global) INTEGER. On exit, <i>lwork</i> is the size of the work buffer. This must be at least $2 * \text{ceil}(\text{ceil}((i-1)/hbl) / \text{lcm}(nprow, npc))$. Here <code>lcm</code> is least common multiple, and $nprow \times npc$ is the logical grid size.

Output Parameters

<i>k</i>	(global) INTEGER. On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $l \leq m \leq i-1$.
<i>buf</i>	(local). REAL for <code>pslasmsub</code> DOUBLE PRECISION for <code>pdlasmsub</code>

Array of size *lwork*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call psllassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

```
call pdlassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

```
call pclassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

```
call pzlassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

Description

The p?lassqroutine returns the values *scl* and *ssq* such that

$$scl^2 * ssq = x_1^2 + \dots + x_n^2 + scale^2 * sumsq,$$

where

$x_i = \text{sub}(X) = X(ix + (jx-1)*m_x + (i-1)*incx)$ for psllassq/pdlassq ,

$x_i = \text{sub}(X) = \text{abs}(X(ix + (jx-1)*m_x + (i-1)*incx))$ for pclassq/pzlassq.

For real routines psllassq/pdlassq the value of *sumsq* is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(x_i)).$$

For complex routines pclassq/pzlassq the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$$1.0 \leq ssq \leq sumsq + 2n$$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i \left(scale, \text{abs}(\text{real}(x_i)), \text{abs}(\text{aimag}(x_i)) \right)$$

For all routines p?lassq values *scale* and *sumsq* must be supplied in *scale* and *sumsq* respectively, and *scale* and *sumsq* are overwritten by *scl* and *ssq* respectively.

All routines p?lassq make only one pass through the vector sub(X).

Input Parameters

<i>n</i>	(global) INTEGER. The length of the distributed vector sub(x).
<i>x</i>	REAL for psllassq DOUBLE PRECISION for pdlassq COMPLEX for pclassq

COMPLEX*16 for pzlassq.

The vector for which a scaled sum of squares is computed:

$x(ix + (jx-1)*m_x + (i - 1)*incx), 1 \leq i \leq n.$

ix (global) INTEGER.

The row index in the global matrix *X* indicating the first row of sub(*X*).

jx (global) INTEGER.

The column index in the global matrix *X* indicating the first column of sub(*X*).

descx (global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix *X*.

incx (global) INTEGER.

The global increment for the elements of *X*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. The argument *incx* must not equal zero.

scale (local).

REAL for pslasq/pclasq

DOUBLE PRECISION for pdlasq/pzlasq.

On entry, the value *scale* in the equation above.

sumsq (local)

REAL for pslasq/pclasq

DOUBLE PRECISION for pdlasq/pzlasq.

On entry, the value *sumsq* in the equation above.

Output Parameters

scale (local).

On exit, *scale* is overwritten with *scl*, the scaling factor for the sum of squares.

sumsq (local).

On exit, *sumsq* is overwritten with the value *sumsq*, the basic sum of squares from which *scl* has been factored out.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

call pslaswp(*direc*, *rowcol*, *n*, *a*, *ia*, *ja*, *desca*, *k1*, *k2*, *ipiv*)

```
call pdlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

Description

The `p?laswp` routine performs a series of row or column interchanges on the distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$. One interchange is initiated for each of rows or columns $k1$ through $k2$ of $\text{sub}(A)$. This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for $k1$ - $k2$ being in the same mb (or nb) block. If you want to pivot a full matrix, use `p?lapiv`.

Input Parameters

<code>direc</code>	(global) CHARACTER. Specifies in which order the permutation is applied: = 'F' - forward, = 'B' - backward.
<code>rowcol</code>	(global) CHARACTER. Specifies if the rows or columns are permuted: = 'R' - rows, = 'C' - columns.
<code>n</code>	(global) INTEGER. If <code>rowcol='R'</code> , the length of the rows of the distributed matrix $A(*, ja:ja+n-1)$ to be permuted; If <code>rowcol='C'</code> , the length of the columns of the distributed matrix $A(ia:ia+n-1, *)$ to be permuted;
<code>a</code>	(local) REAL for <code>pslaswp</code> DOUBLE PRECISION for <code>pdlaswp</code> COMPLEX for <code>pclaswp</code> COMPLEX*16 for <code>pzlaswp</code> . Pointer into the local memory to an array of size $(lld_a, *)$. On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.
<code>ia</code>	(global) INTEGER. The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
<code>ja</code>	(global) INTEGER. The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
<code>desca</code>	(global and local) INTEGER array of size $dlen_$.

	The array descriptor for the distributed matrix A .
$k1$	(global) INTEGER. The first element of $ipiv$ for which a row or column interchange will be done.
$k2$	(global) INTEGER. The last element of $ipiv$ for which a row or column interchange will be done.
$ipiv$	(local) INTEGER. Array of size $LOCr(m_a)+mb_a$ for row pivoting and $LOCr(n_a)+nb_a$ for column pivoting. This array is tied to the matrix A , $ipiv(k)=l$ implies rows (or columns) k and l are to be interchanged.

Output Parameters

A	(local) REAL for pslaswp DOUBLE PRECISION for pdlaswp COMPLEX for pclaswp COMPLEX*16 for pzlaswp. On exit, the permuted distributed matrix.
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

```
val = pslatra(n, a, ia, ja, desca)
```

```
val = pdlatra(n, a, ia, ja, desca)
```

```
val = pclatra(n, a, ia, ja, desca)
```

```
val = pzlatra(n, a, ia, ja, desca)
```

Description

This function computes the trace of an n -by- n distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$. The result is left on every process of the grid.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
a	(local). REAL for pslatra

DOUBLE PRECISION for pdlatra

COMPLEX for pclatra

COMPLEX*16 for pzlatra.

Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$ containing the local pieces of the distributed matrix, the trace of which is to be computed.

ia, ja (global) INTEGER. The row and column indices respectively in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

desca (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

val The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call pslatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

```
call pdlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

```
call pclatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

```
call pzlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

Description

The p?latrd routine reduces nb rows and columns of a real symmetric or complex Hermitian matrix sub(A) = $A(ia:ia+n-1, ja:ja+n-1)$ to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation $Q'^*sub(A)*Q$, and returns the matrices V and W , which are needed to apply the transformation to the unreduced part of sub(A).

If $uplo = U$, p?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if $uplo = L$, p?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by [p?sytrd/p?hetrd](#).

Input Parameters

uplo (global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored:

	= 'U': Upper triangular = L: Lower triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
<i>nb</i>	(global) INTEGER. The number of rows and columns to be reduced.
<i>a</i>	REAL for <code>pslatrd</code> DOUBLE PRECISION for <code>pdlatrd</code> COMPLEX for <code>pclatrd</code> COMPLEX*16 for <code>pzlatrd</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix $\text{sub}(A)$. If $uplo = U$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = L$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia</i>	(global) INTEGER. The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>iw</i>	(global) INTEGER. The row index in the global matrix W indicating the first row of $\text{sub}(W)$.
<i>jwt</i>	(global) INTEGER. The column index in the global matrix W indicating the first column of $\text{sub}(W)$.
<i>descw</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix W .
<i>work</i>	(local) REAL for <code>pslatrd</code> DOUBLE PRECISION for <code>pdlatrd</code>

COMPLEX for pclatrd
 COMPLEX*16 for pzlatrd.
 Workspace array of size nb_a .

Output Parameters

- a* (local)
 On exit, if $uplo = 'U'$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $sub(A)$; the elements above the diagonal with the array tau represent the orthogonal/unitary matrix Q as a product of elementary reflectors;
 if $uplo = 'L'$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $sub(A)$; the elements below the diagonal with the array tau represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
- d* (local)
 REAL for pslatrd/pclatrd
 DOUBLE PRECISION for pdlatrd/pzlatrd.
 Array of size $LOCc(ja+n-1)$.
 The diagonal elements of the tridiagonal matrix T : $d(i) = a(i,i)$. d is tied to the distributed matrix A .
- e* (local)
 REAL for pslatrd/pclatrd
 DOUBLE PRECISION for pdlatrd/pzlatrd.
 Array of size $LOCc(ja+n-1)$ if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.
 The off-diagonal elements of the tridiagonal matrix T :
 $e(i) = a(i, i + 1)$ if $uplo = 'U'$,
 $e(i) = a(i + 1, i)$ if $uplo = 'L'$.
 e is tied to the distributed matrix A .
- tau* (local)
 REAL for pslatrd
 DOUBLE PRECISION for pdlatrd
 COMPLEX for pclatrd
 COMPLEX*16 for pzlatrd.
 Array of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A .
- w* (local)
 REAL for pslatrd
 DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Pointer into the local memory to an array of size $lld_wby \cdot nb_w$. This array contains the local pieces of the n -by- nb_w matrix w required to update the unreduced part of $sub(A)$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-1, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1: ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank- $2k$ update of the form:

$$sub(A) := sub(A) - v w^T - w v^T.$$

The contents of a on exit are illustrated by the following examples with

$n = 5$ and $nb = 2$:

if $uplo='U'$:	if $uplo='L'$:
$\begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix}$	$\begin{bmatrix} d \\ 1 & d \\ v_1 & 1 & a \\ v_1 & v_2 & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call pslatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)
```

```
call pdlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)
```

```
call pclatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)
```

```
call pzlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)
```

Description

The p?latrs routine solves a triangular system of equations $Ax = sb$, $A^T x = sb$ or $A^H x = sb$, where s is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to Ax . = 'N': Solve $Ax = s*b$ (no transpose) = 'T': Solve $A^T x = s*b$ (transpose) = 'C': Solve $A^H x = s*b$ (conjugate transpose), where s - is a scale factor
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular = 'U': Unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER.

	The order of the matrix A . $n \geq 0$
<i>a</i>	<p>REAL for pslatrs/pclatrs</p> <p>DOUBLE PRECISION for pdlatrs/pzlatrs</p> <p>Array of size ld by n. Contains the triangular matrix A.</p> <p>If $uplo = U$, the leading n-by-n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced.</p> <p>If $uplo = 'L'$, the leading n-by-n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.</p> <p>If $diag = 'U'$, the diagonal elements of a are also not referenced and are assumed to be 1.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>x</i>	<p>REAL for pslatrs/pclatrs</p> <p>DOUBLE PRECISION for pdlatrs/pzlatrs</p> <p>Array of size n. On entry, the right hand side b of the triangular system.</p>
<i>ix</i>	(global) INTEGER. The row index in the global matrix X indicating the first row of sub(x).
<i>jx</i>	<p>(global) INTEGER.</p> <p>The column index in the global matrix X indicating the first column of sub(X).</p>
<i>descx</i>	<p>(global and local) INTEGER.</p> <p>Array of size $dlen_$. The array descriptor for the distributed matrix X.</p>
<i>cnorm</i>	<p>REAL for pslatrs/pclatrs</p> <p>DOUBLE PRECISION for pdlatrs/pzlatrs.</p> <p>Array of size n. If $normin = 'Y'$, $cnorm$ is an input argument and $cnorm(j)$ contains the norm of the off-diagonal part of the j-th column of A. If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or $'C'$, $cnorm(j)$ must be greater than or equal to the 1-norm.</p>
<i>work</i>	<p>(local).</p> <p>REAL for pslatrs</p> <p>DOUBLE PRECISION for pdlatrs</p> <p>COMPLEX for pclatrs</p> <p>COMPLEX*16 for pzlatrs.</p>

Temporary workspace.

Output Parameters

x	On exit, x is overwritten by the solution vector x .
$scale$	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs. Array of size $ldaby$ n . The scaling factor s for the triangular system as described above. If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $Ax = 0$.
$cnorm$	If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j -th column of A .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
call pslatz(m, n, l, a, ia, ja, desca, tau, work)
call pdlatrz(m, n, l, a, ia, ja, desca, tau, work)
call pclatz(m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz(m, n, l, a, ia, ja, desca, tau, work)
```

Description

The p?latrz routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = [A(ia:ia+m-1, ja:ja+m-1)A(ia:ia+m-1, ja+n-l:ja+n-1)]$ to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix $\text{sub}(A)$ is factored as

$$\text{sub}(A) = (R \ 0) * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$. $m \geq 0$.
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$. $n \geq 0$.
l	(global) INTEGER.

The number of columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. $l > 0$.

a

(local)

REAL for `pslatrz`

DOUBLE PRECISION for `pdlatz`

COMPLEX for `pclatz`

COMPLEX*16 for `pzlatrz`.

Pointer into the local memory to an array of size $(lld_a, LOCc(ja+n-1))$. On entry, the local pieces of the m -by- n distributed matrix $\text{sub}(A)$, which is to be factored.

ia

(global) INTEGER.

The row index in the global matrix A indicating the first row of $\text{sub}(A)$.

ja

(global) INTEGER.

The column index in the global matrix A indicating the first column of $\text{sub}(A)$.

desca

(global and local) INTEGER array of size *dlen_*.

The array descriptor for the distributed matrix A .

work

(local)

REAL for `pslatrz`

DOUBLE PRECISION for `pdlatz`

COMPLEX for `pclatz`

COMPLEX*16 for `pzlatrz`.

Workspace array of size *lwork*.

$lwork \geq nq0 + \max(1, mp0)$, where

$iroff = \text{mod}(ia-1, mb_a)$,

$icoff = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprw)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcw)$,

$mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprw)$,

$nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcw)$,

`numroc`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; *myrow*, *mycol*, *nprw*, and *npcw* can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

<i>a</i>	On exit, the leading m -by- m upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix R , and elements $n-l+1$ to n of the first m rows of $\text{sub}(A)$, with the array <i>tau</i> , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.
<i>tau</i>	(local) REAL for pslatz DOUBLE PRECISION for pdlatrz COMPLEX for pclatz COMPLEX*16 for pzlatrz. Array of size $LOCr(ja+m-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A .

Application Notes

The factorization is obtained by Householder's method. The k -th transformation matrix, $Z(k)$, which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the $(m - k + 1)$ -th row of $\text{sub}(A)$, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - \tau u(k) u(k)', \quad u(k) = \begin{bmatrix} I \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and $z(k)$ is an $(n-m)$ -element vector. *tau* and $z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar *tau* is returned in the k -th element of *tau* and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $z(k)$ are in $A(k, m+1), \dots, A(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$.

Z is given by

$$Z = Z(1)Z(2)\dots Z(m).$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lauu2

Computes the product U^*U' or L'^*L , where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

```
call pslauu2(uplo, n, a, ia, ja, desca)
```

```
call pdlauu2(uplo, n, a, ia, ja, desca)
```

```
call pclauu2(uplo, n, a, ia, ja, desca)
```

call pzlaau2(uplo, n, a, ia, ja, desca)

Description

The pzlaau2 routine computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the distributed matrix

$sub(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $sub(A)$.

If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $sub(A)$.

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#). No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the triangular factor stored in the <i>matrix</i> $sub(A)$ is upper or lower triangular: = U: upper triangular = L: lower triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.
<i>a</i>	(local) REAL for pslau2 DOUBLE PRECISION for pdlau2 COMPLEX for pclau2 COMPLEX*16 for pzlaau2. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, the local pieces of the triangular factor U or L .
<i>ia</i>	(global) INTEGER. The row index in the global matrix A indicating the first row of $sub(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global matrix A indicating the first column of $sub(A)$.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

<i>a</i>	(local)
----------	---------

On exit, if $uplo = 'U'$, the upper triangle of the distributed matrix $sub(A)$ is overwritten with the upper triangle of the product $U*U'$; if $uplo = 'L'$, the lower triangle of $sub(A)$ is overwritten with the lower triangle of the product $L'*L$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lauum

*Computes the product $U*U'$ or $L'*L$, where U and L are upper or lower triangular matrices.*

Syntax

```
call pslauum(uplo, n, a, ia, ja, desca)
```

```
call pdlauum(uplo, n, a, ia, ja, desca)
```

```
call pclauum(uplo, n, a, ia, ja, desca)
```

```
call pzlauum(uplo, n, a, ia, ja, desca)
```

Description

The `p?lauum` routine computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the matrix $sub(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $sub(A)$. If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $sub(A)$.

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the triangular factor stored in the matrix $sub(A)$ is upper or lower triangular: = 'U': upper triangular = 'L': lower triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.
<i>a</i>	(local) REAL for pslauum DOUBLE PRECISION for pdlauum COMPLEX for pclauum COMPLEX*16 for pzlauum. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, the local pieces of the triangular factor U or L .
<i>ia</i>	(global) INTEGER.

	The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
ja	(global) INTEGER. The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a	(local) On exit, if $uplo = 'U'$, the upper triangle of the distributed matrix $\text{sub}(A)$ is overwritten with the upper triangle of the product $U*U'$; if $uplo = 'L'$, the lower triangle of $\text{sub}(A)$ is overwritten with the lower triangle of the product $L'*L$.
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lawil

Forms the Wilkinson transform.

Syntax

```
call pslawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
```

Description

The p?lawil routine gets the transform given by $h44$, $h33$, and $h43h34$ into v starting at row m .

Input Parameters

ii	(global) INTEGER. Row owner of $h(m+2, m+2)$.
jj	(global) INTEGER. Column owner of $h(m+2, m+2)$.
m	(global) INTEGER. On entry, the location from where the transform starts (row m). Unchanged on exit.
a	(global) REAL for pslawil DOUBLE PRECISION for pdlawil Array of size $(lld_a, *)$. On entry, the Hessenberg matrix. Unchanged on exit.
$desca$	(global and local) INTEGER

Array of size $dlen_$. The array descriptor for the distributed matrix A . Unchanged on exit.

h43h34

(global)

REAL for `pslawil`

DOUBLE PRECISION for `pdlawil`

These three values are for the double shift QR iteration. Unchanged on exit.

Output Parameters

v

(global)

REAL for `pslawil`

DOUBLE PRECISION for `pdlawil`

Array of size 3 that contains the transform on output.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by `p?geqlf` (unblocked algorithm).

Syntax

```
call psorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pdorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pcung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pzung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?org2l/p?ung2l` routine generates an m -by- n real/complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$Q = H(k) * \dots * H(2) * H(1)$ as returned by `p?geqlf`.

Input Parameters

m

(global) INTEGER.

The number of rows in the distributed submatrix Q . $m \geq 0$.

n

(global) INTEGER.

The number of columns in the distributed submatrix Q . $m \geq n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.

a REAL for psorg21
DOUBLE PRECISION for pdorg21
COMPLEX for pcung21
COMPLEX*16 for pzung21.
Pointer into the local memory to an array of size (*lld_a*, *LOCc(ja+n-1)*).
On entry, the *j*-th column must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-k$, as returned by `p?geqlf` in the *k* columns of its *distributed matrix* argument *A* (*ia* : *, *ja+n-k* : *ja+n-1*).

ia (global) INTEGER.
The row index in the global matrix *A* indicating the first row of sub(*A*).

ja (global) INTEGER.
The column index in the global matrix *A* indicating the first column of sub(*A*).

desca (global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

tau (local)
REAL for psorg21
DOUBLE PRECISION for pdorg21
COMPLEX for pcung21
COMPLEX*16 for pzung21.
Array of size *LOCc(ja+n-1)*.
tau(j) contains the scalar factor of the elementary reflector $H(j)$, as returned by `p?geqlf`.

work (local)
REAL for psorg21
DOUBLE PRECISION for pdorg21
COMPLEX for pcung21
COMPLEX*16 for pzung21.
Workspace array of size *lwork*.

lwork (local or global) INTEGER.
The size of the array *work*.
lwork is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where

$$iroffa = \text{mod}(ia-1, mb_a),$$

$$icoffa = \text{mod}(ja-1, nb_a),$$

$$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$$

```
iacol = indxg2p(ja, nb_a, mycol, csrc_a, npc1),
mpa0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nqa0 = numroc(n+icoffa, nb_a, mycol, iacol, npc1).
```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<code>work</code>	On exit, <code>work(1)</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local)INTEGER. = 0: successful exit < 0: if the i -th argument is an array and the j -th entry had an illegal value, then <code>info = -(i*100 + j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by `p?geqrf` (unblocked algorithm).

Syntax

```
call psorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?org2r/p?ung2r` routine generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m :

$$Q = H(1)*H(2)*...*H(k)$$

as returned by `p?geqrf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix Q . $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix Q . $m \geq n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
<i>a</i>	REAL for psorg2r DOUBLE PRECISION for pdorg2r COMPLEX for pcung2r COMPLEX*16 for pzung2r. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$ On entry, the j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its <i>distributed matrix</i> argument A ($ia:*$, $ja:ja+k-1$).
<i>ia</i>	(global) INTEGER. The row index in the global matrix A indicating the first row of sub(A).
<i>ja</i>	(global) INTEGER. The column index in the global matrix A indicating the first column of sub(A).
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for psorg2r DOUBLE PRECISION for pdorg2r COMPLEX for pcung2r COMPLEX*16 for pzung2r. Array of size $LOCC(ja+k-1)$. $tau(j)$ contains the scalar factor of the elementary reflector $H(j)$, as returned by <code>p?geqrf</code> . This array is tied to the distributed matrix A .
<i>work</i>	(local) REAL for psorg2r DOUBLE PRECISION for pdorg2r COMPLEX for pcung2r COMPLEX*16 for pzung2r.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.
The size of the array *work*.
lwork is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where

$$iroffa = \text{mod}(ia-1, mb_a), \quad icoffa = \text{mod}(ja-1, nb_a),$$

$$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$$

$$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot),$$

$$mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow),$$

$$nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot).$$

`indxg2p` and `numroc` are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcot* can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, this array contains the local pieces of the *m*-by-*n* distributed matrix *Q*.

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100 + *j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
call psorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The `p?orgl2/p?ungl2` routine generates a m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) \dots H(2) H(1)$ (for real flavors),

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ (for complex flavors) as returned by `p?gelqf`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix Q . $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix Q . $n \geq m \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
<i>a</i>	REAL for <code>psorgl2</code> DOUBLE PRECISION for <code>pdorgl2</code> COMPLEX for <code>pcungl2</code> COMPLEX*16 for <code>pzungl2</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the k rows of its <i>distributed matrix</i> argument $A(ia:ia+k-1, ja:*)$.
<i>ia</i>	(global) INTEGER. The row index in the global matrix A indicating the first row of $sub(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global matrix A indicating the first column of $sub(A)$.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for <code>psorgl2</code> DOUBLE PRECISION for <code>pdorgl2</code> COMPLEX for <code>pcungl2</code> COMPLEX*16 for <code>pzungl2</code> . Array of size $LOCr(ja+k-1)$. $tau(j)$ contains the scalar factor of the elementary reflectors $H(j)$, as returned by <code>p?gelqf</code> . This array is tied to the distributed matrix A .

<i>WORK</i>	(local) REAL for <i>psorgl2</i> DOUBLE PRECISION for <i>pdorgl2</i> COMPLEX for <i>pcungl2</i> COMPLEX*16 for <i>pzungl2</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npc1),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npc1).$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	On exit, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgr2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

```
call psorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

Description

The p?orgr2/p?ungr2 routine generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1)*H(2)*...*H(k)$ (for real flavors);

$Q = (H(1))^H*(H(2))^H*...*(H(k))^H$ (for complex flavors) as returned by p?gerqf.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed submatrix Q . $m \geq 0$.
n	(global) INTEGER. The number of columns in the distributed submatrix Q . $n \geq m \geq 0$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
a	REAL for psorgr2 DOUBLE PRECISION for pdorgr2 COMPLEX for pcungr2 COMPLEX*16 for pzungr2. Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$. On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia+m-k:ia+m-1, ja:*)$.
ia	(global) INTEGER. The row index in the global matrix A indicating the first row of sub(A).
ja	(global) INTEGER. The column index in the global matrix A indicating the first column of sub(A).

<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for <i>psorgr2</i> DOUBLE PRECISION for <i>pdorgr2</i> COMPLEX for <i>pcungr2</i> COMPLEX*16 for <i>pzungr2</i> . Array of size $LOCr(ja+m-1)$. <i>tau(j)</i> contains the scalar factor of the elementary reflectors $H(j)$, as returned by p?gerqf . This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <i>psorgr2</i> DOUBLE PRECISION for <i>pdorgr2</i> COMPLEX for <i>pcungr2</i> COMPLEX*16 for <i>pzungr2</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $irowfa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$, $irow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcil)$, $mpa0 = \text{numroc}(m+irowfa, mb_a, myrow, irow, nprow)$, $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcil)$. <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcil</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla .

Output Parameters

<i>a</i>	On exit, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value,

then *info* = - (*i**100 + *j*),
 if the *i*-th argument is a scalar and had an illegal value,
 then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

```
call psorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pcunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Description

The p?orm2l/p?unm2l routine overwrites the general real/complex *m*-by-*n* distributed matrix sub (*C*)=*C*(*ic:ic+m-1,jc:jc+n-1*) with

$Q \cdot \text{sub}(C)$ if *side* = 'L' and *trans* = 'N', or

$Q^T \cdot \text{sub}(C)$ / $Q^H \cdot \text{sub}(C)$ if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

$\text{sub}(C) \cdot Q$ if *side* = 'R' and *trans* = 'N', or

$\text{sub}(C) \cdot Q^T$ / $\text{sub}(C) \cdot Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

where *Q* is a real orthogonal or complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(k) \cdot \dots \cdot H(2) \cdot H(1)$ as returned by p?geqlf . *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

side (global) CHARACTER.
 = 'L': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the left,
 = 'R': apply *Q* or Q^T for real flavors (Q^H for complex flavors) from the right.

trans (global) CHARACTER.
 = 'N': apply *Q* (no transpose)

	<p>= 'T': apply Q^T (transpose, for real flavors)</p> <p>= 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	<p>(global) INTEGER.</p> <p>The number of rows in the distributed matrix sub(C). $m \geq 0$.</p>
<i>n</i>	<p>(global) INTEGER.</p> <p>The number of columns in the distributed matrix sub(C). $n \geq 0$.</p>
<i>k</i>	<p>(global) INTEGER.</p> <p>The number of elementary reflectors whose product defines the matrix Q.</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$;</p> <p>if <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>REAL for psorm21</p> <p>DOUBLE PRECISION for pdorm21</p> <p>COMPLEX for pcunm21</p> <p>COMPLEX*16 for pzunm21.</p> <p>Pointer into the local memory to an array of size $(lld_a, LOCc(ja+k-1))$.</p> <p>On entry, the j-th row must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.</p> <p>If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$,</p> <p>if <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$.</p>
<i>ia</i>	<p>(global) INTEGER.</p> <p>The row index in the global matrix A indicating the first row of sub(A).</p>
<i>ja</i>	<p>(global) INTEGER.</p> <p>The column index in the global matrix A indicating the first column of sub(A).</p>
<i>desca</i>	<p>(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A.</p>
<i>tau</i>	<p>(local)</p> <p>REAL for psorm21</p> <p>DOUBLE PRECISION for pdorm21</p> <p>COMPLEX for pcunm21</p> <p>COMPLEX*16 for pzunm21.</p>

Array of size $LOCc(ja+n-1)$. $\tau(j)$ contains the scalar factor of the elementary reflector $H(j)$, as returned by `p?geqlf`. This array is tied to the distributed matrix A .

c

(local)

REAL for `psorm21`
 DOUBLE PRECISION for `pdorm21`
 COMPLEX for `pcunm21`
 COMPLEX*16 for `pzunm21`.

Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$. On entry, the local pieces of the distributed matrix sub(C).

ic

(global) INTEGER.

The row index in the global matrix C indicating the first row of sub(C).

jc

(global) INTEGER.

The column index in the global matrix C indicating the first column of sub(C).

desc

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local)

REAL for `psorm21`
 DOUBLE PRECISION for `pdorm21`
 COMPLEX for `pcunm21`
 COMPLEX*16 for `pzunm21`.

Workspace array of size $lwork$.

On exit, `work(1)` returns the minimal and optimal $lwork$.

lwork

(local or global) INTEGER.

The size of the array $work$.

$lwork$ is local input and must be at least

if $side = 'L'$, $lwork \geq mpc0 + \max(1, nqc0)$,

if $side = 'R'$, $lwork \geq nqc0 + \max(\max(1, mpc0), \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, npc0l), nb_a, 0, 0, lcmq))$,

where

$lcmq = lcm/npc0l$,

$lcm = iclm(nprow, npc0l)$,

$iroffc = \text{mod}(ic-1, mb_c)$,

$icoffc = \text{mod}(jc-1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,

$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0l)$,

$Mqc0 = \text{numroc}(m+icoffc, nb_c, mycol, icrow, nprow),$

$Npc0 = \text{numroc}(n+iroffc, mb_c, myrow, iccol, npc0),$

$ilcm, \text{indxg2p},$ and numroc are ScaLAPACK tool functions; $myrow, mycol, nprow,$ and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1,$ then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c	On exit, c is overwritten by $Q*\text{sub}(C),$ or $Q^T*\text{sub}(C)/ Q^H*\text{sub}(C),$ or $\text{sub}(C)*Q,$ or $\text{sub}(C)*Q^T / \text{sub}(C)*Q^H$
$work$	On exit, $work(1)$ returns the minimal and optimal $lwork.$
$info$	(local) INTEGER. = 0: successful exit < 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100 + j),$ if the i -th argument is a scalar and had an illegal value, then $info = -i.$

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L',$ ($mb_a.eq.mb_c .AND. iroffa.eq.iroffc .AND. iarow.eq.icrow$)

If $side = 'R',$ ($mb_a.eq.nb_c .AND. iroffa.eq.iroffc$).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?

geqrf (unblocked algorithm).

Syntax

call `psorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

call `pdorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

call `pcunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

call `pzunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)`

Description

The `p?orm2r/p?unm2r` routine overwrites the general real/complex m -by- n distributed matrix sub(C)= $C(ic:ic+m-1, jc:jc+n-1)$ with

$Q*\text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T*\text{sub}(C) / Q^H*\text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C)*Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C)*Q^T / \text{sub}(C)*Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k)*\dots*H(2)*H(1)$ as returned by `p?geqrf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER. = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C). $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C). $n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for <code>psorm2r</code> DOUBLE PRECISION for <code>pdorm2r</code> COMPLEX for <code>pcunm2r</code> COMPLEX*16 for <code>pzunm2r</code> . Pointer into the local memory to an array of size $(lld_a, LOCC(ja+k-1))$.

On entry, the j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by `p?geqrf` in the k columns of its distributed matrix argument A ($ia:*$, $ja:ja+k-1$). The argument A ($ia:*$, $ja:ja+k-1$) is modified by the routine but restored on exit.

If $side = 'L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$,

if $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$.

<i>ia</i>	(global) INTEGER. The row index in the global matrix A indicating the first row of $sub(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global matrix A indicating the first column of $sub(A)$.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for <code>psorm2r</code> DOUBLE PRECISION for <code>pdorm2r</code> COMPLEX for <code>pcunm2r</code> COMPLEX*16 for <code>pzunm2r</code> . Array of size $LOCc(ja+k-1)$. $tau(j)$ contains the scalar factor of the elementary reflector $H(j)$, as returned by <code>p?geqrf</code> . This array is tied to the distributed matrix A .
<i>c</i>	(local) REAL for <code>psorm2r</code> DOUBLE PRECISION for <code>pdorm2r</code> COMPLEX for <code>pcunm2r</code> COMPLEX*16 for <code>pzunm2r</code> . Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$. On entry, the local pieces of the distributed matrix $sub(C)$.
<i>ic</i>	(global) INTEGER. The row index in the global matrix C indicating the first row of $sub(C)$.
<i>jc</i>	(global) INTEGER. The column index in the global matrix C indicating the first column of $sub(C)$.
<i>descc</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .
<i>work</i>	(local)

REAL for psorm2r
 DOUBLE PRECISION for pdorm2r
 COMPLEX for pcunm2r
 COMPLEX*16 for pzunm2r.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if *side* = 'L', $lwork \geq mpc0 + \max(1, nqc0)$,

if *side* = 'R', $lwork \geq nqc0 + \max(\max(1, mpc0), \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, npc0l), nb_a, 0, 0, lcmq))$,

where

$lcmq = lcm / npc0l$,

$lcm = iclm(nprow, npc0l)$,

$iroffc = \text{mod}(ic-1, mb_c)$,

$icoffc = \text{mod}(jc-1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,

$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0l)$,

$Mqc0 = \text{numroc}(m + icoffc, nb_c, mycol, icrow, nprow)$,

$Npc0 = \text{numroc}(n + iroffc, mb_c, myrow, iccol, npc0l)$,

iclm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npc0l* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

c

On exit, *c* is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(i \cdot 100 + j)$,

if the *i*-th argument is a scalar and had an illegal value,

then $info = -i$.

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, ($mb_a.eq.mb_c$.AND. $iroffa.eq.iroffc$.AND. $iarow.eq.icrow$).

If $side = 'R'$, ($mb_a.eq.nb_c$.AND. $iroffa.eq.iroffc$).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
call psorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pcunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Description

The p?orml2/p?unml2 routine overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1, jc:jc+n-1)$ with

$Q*\text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T*\text{sub}(C) / Q^H*\text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C)*Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C)*Q^T / \text{sub}(C)*Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k)*\dots*H(2)*H(1)$ (for real flavors)

$Q = (H(k))^H*\dots*(H(2))^H*(H(1))^H$ (for complex flavors)

as returned by p?gelqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global) CHARACTER.

= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,

	<p>= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.</p>
<i>trans</i>	<p>(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	<p>(global) INTEGER. The number of rows in the distributed matrix sub(C). $m \geq 0$.</p>
<i>n</i>	<p>(global) INTEGER. The number of columns in the distributed matrix sub(C). $n \geq 0$.</p>
<i>k</i>	<p>(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q. If <i>side</i> = 'L', $m \geq k \geq 0$; if <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>(local) REAL for psorml2 DOUBLE PRECISION for pdorml2 COMPLEX for pcunml2 COMPLEX*16 for pzunml2. Pointer into the local memory to an array of size (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>m</i>-1)) if <i>side</i>='L', (<i>lld_a</i>, <i>LOCc</i>(<i>ja</i>+<i>n</i>-1)) if <i>side</i>='R', where <i>lld_a</i> $\geq \max(1, \text{LOCr}(\textit{ia}+\textit{k}-1))$. On entry, the <i>i</i>-th row must contain the vector that defines the elementary reflector $H(i)$, $\textit{ia} \leq i \leq \textit{ia}+\textit{k}-1$, as returned by p?gelqf in the <i>k</i> rows of its distributed matrix argument $A(\textit{ia}:\textit{ia}+\textit{k}-1, \textit{ja}:\ast)$. The argument $A(\textit{ia}:\textit{ia}+\textit{k}-1, \textit{ja}:\ast)$ is modified by the routine but restored on exit.</p>
<i>ia</i>	<p>(global) INTEGER. The row index in the global matrix A indicating the first row of sub(A).</p>
<i>ja</i>	<p>(global) INTEGER. The column index in the global matrix A indicating the first column of sub(A).</p>
<i>desca</i>	<p>(global and local) INTEGER array of size <i>dlen_</i>. The array descriptor for the distributed matrix A.</p>
<i>tau</i>	<p>(local) REAL for psorml2</p>

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Array of size $LOCc(ia+k-1)$. $\tau(i)$ contains the scalar factor of the elementary reflector $H(i)$, as returned by [p?ge1qf](#). This array is tied to the distributed matrix A .

c

(local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$. On entry, the local pieces of the distributed matrix sub(C).

ic

(global) INTEGER.

The row index in the global matrix C indicating the first row of sub(C).

jc

(global) INTEGER.

The column index in the global matrix C indicating the first column of sub(C).

desc

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Workspace array of size *lwork*.

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if $side = 'L'$, $lwork \geq mqc0 + \max(\max(1, npc0), \text{numroc}(\text{numroc}(m + icoffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcm p))$,

if $side = 'R'$, $lwork \geq npc0 + \max(1, mqc0)$,

where

$lcmp = lcm / nprow$,

$lcm = iclm(nprow, npc0l)$,

$iroffc = \text{mod}(ic-1, mb_c)$,

$icoffc = \text{mod}(jc-1, nb_c)$,

```

icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
Mpc0 = numroc(m+icoffc, mb_c, mycol, icrow, nprow),
Nqc0 = numroc(n+iroffc, nb_c, myrow, iccol, npcol),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the subroutine
blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a*.eq.*mb_c* .AND. *icoffa*.eq.*iroffc*)

If *side* = 'R', (*nb_a*.eq.*nb_c* .AND. *icoffa*.eq.*icoffc* .AND. *iacol*.eq.*iccol*).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

```
call psormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pcunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

Description

The `pcunmr2`/`pzunmr2` routine overwrites the general real/complex m -by- n distributed matrix sub(C)= $C(ic:ic+m-1, jc:jc+n-1)$ with

$Q \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) \cdot Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) \cdot H(2) \cdot \dots \cdot H(k)$ (for real flavors)

$Q = (H(1))^H \cdot (H(2))^H \cdot \dots \cdot (H(k))^H$ (for complex flavors)

as returned by `p?gerqf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER. = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(C). $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(C). $n \geq 0$.
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) REAL for <code>psormr2</code>

DOUBLE PRECISION for pdormr2
 COMPLEX for pcunmr2
 COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array of size

$(lld_a, LOCc(ja+m-1))$ if $side='L'$,
 $(lld_a, LOCc(ja+n-1))$ if $side='R'$,

where $lld_a \geq \max(1, LOCr(ia+k-1))$.

On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by [p?gerqf](#) in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

The argument $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia (global) INTEGER.

The row index in the global matrix A indicating the first row of sub(A).

ja (global) INTEGER.

The column index in the global matrix A indicating the first column of sub(A).

desca (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

tau (local)

REAL for psormr2
 DOUBLE PRECISION for pdormr2
 COMPLEX for pcunmr2
 COMPLEX*16 for pzunmr2.

Array of size $LOCc(ia+k-1)$. $tau(j)$ contains the scalar factor of the elementary reflector $H(j)$, as returned by [p?gerqf](#). This array is tied to the distributed matrix A .

c (local)

REAL for psormr2
 DOUBLE PRECISION for pdormr2
 COMPLEX for pcunmr2
 COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array of size $(lld_c, LOCc(jc+n-1))$.
 On entry, the local pieces of the distributed matrix sub (C).

ic (global) INTEGER.

The row index in the global matrix C indicating the first row of sub(C).

jc (global) INTEGER.

The column index in the global matrix C indicating the first column of $\text{sub}(C)$.

desc (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix C .

work (local)

REAL for `psormr2`

DOUBLE PRECISION for `pdormr2`

COMPLEX for `pcunmr2`

COMPLEX*16 for `pzunmr2`.

Workspace array of size *lwork*.

lwork (local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least

if $side = 'L'$, $lwork \geq mpc0 + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcm))$,

if $side = 'R'$, $lwork \geq nqc0 + \max(1, mpc0)$,

where $lcmp = lcm/nprow$,

$lcm = iclm(nprow, npc0)$,

$iroffc = \text{mod}(ic-1, mb_c)$,

$icoffc = \text{mod}(jc-1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,

$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0)$,

$Mpc0 = \text{numroc}(m + iroffc, mb_c, myrow, icrow, nprow)$,

$Nqc0 = \text{numroc}(n + icoffc, nb_c, mycol, iccol, npc0)$,

$iclm$, indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npc0$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c On exit, *c* is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C)$ / $Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T$ / $\text{sub}(C) \cdot Q^H$

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value,

then $info = -(i*100 + j)$,
 if the i -th argument is a scalar and had an illegal value,
 then $info = -i$.

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, ($nb_a.eq.mb_c$.AND. $icoffa.eq.iroffc$).

If $side = 'R'$, ($nb_a.eq.nb_c$.AND. $icoffa.eq.icoffc$.AND. $iacol.eq.iccol$).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrsv

Solves a single triangular linear system via [frontsolve](#) or [backsolve](#) where the triangular matrix is a factor of a banded matrix computed by [p?pbtrf](#).

Syntax

```
call pspbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pdpbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pcpbtrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

```
call pzpbttrsv(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

Description

The `p?pbtrsv` routine solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code [p?pbtrf](#) and is stored in $A(1:n, ja:ja+n-1)$ and af . The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$.

The routine [p?pbtrf](#) must be called first.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice

effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <i>uplo</i> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$; If <i>trans</i> = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$. If <i>trans</i> = 'C' for complex flavors, solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>bw</i>	(global) INTEGER. The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>a</i>	(local) REAL for <i>pspbtrsv</i> DOUBLE PRECISION for <i>pdpbtrsv</i> COMPLEX for <i>pcpbtrsv</i> COMPLEX*16 for <i>pzpbtrsv</i> . Pointer into the local memory to an array with the first size $lld_a \geq (bw + 1)$, stored in <i>desca</i> . On entry, this array contains the local pieces of the n -by- n symmetric banded distributed Cholesky factor L or $L^T * A(1:n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.

<i>ja</i>	(global) INTEGER. The index in the global in the global matrix <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If 1D type (<i>dtype_a</i> = 501), then <i>dlen</i> ≥ 7; If 2D type (<i>dtype_a</i> = 1), then <i>dlen</i> ≥ 9. Contains information on mapping of <i>A</i> to memory. (See ScaLAPACK manual for full description and options.)
<i>b</i>	(local) REAL for <i>pspbtrsv</i> DOUBLE PRECISION for <i>pdpbtrsv</i> COMPLEX for <i>pcpbtrsv</i> COMPLEX*16 for <i>pzpbtrsv</i> . Pointer into the local memory to an array of local lead size <i>lld_b</i> ≥ <i>nb</i> . On entry, this array contains the local pieces of the right hand sides <i>B(jb:jb+n-1, 1:nrhs)</i> .
<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . If 1D type (<i>dtype_b</i> = 502), then <i>dlen</i> ≥ 7; If 2D type (<i>dtype_b</i> = 1), then <i>dlen</i> ≥ 9. Contains information on mapping of <i>B</i> to memory. Please, see ScaLAPACK manual for full description and options.
<i>laf</i>	(local) INTEGER. The size of user-input auxiliary fill-in space <i>af</i> . Must be <i>laf</i> ≥ (<i>nb</i> +2* <i>bw</i>)* <i>bw</i> . If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) REAL for <i>pspbtrsv</i> DOUBLE PRECISION for <i>pdpbtrsv</i> COMPLEX for <i>pcpbtrsv</i> COMPLEX*16 for <i>pzpbtrsv</i> . The array <i>work</i> is a temporary workspace array of size <i>lwork</i> . This space may be overwritten in between calls to routines.

lwork (local or global) INTEGER. The size of the user-input workspace *work*, must be at least $lwork \geq bw * nrhs$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af (local)
 REAL for pspbtrsv
 DOUBLE PRECISION for pdpbtrsv
 COMPLEX for pcpbtrsv
 COMPLEX*16 for pzpbttrsv.
 The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine [p?pbtrf](#) and is stored in *af*. If a linear system is to be solved using [p?pbtrs](#) after the factorization routine, *af* must not be altered after the factorization.

b On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1) On exit, *work*(1) contains the minimum value of *lwork*.

info (local) INTEGER.
 = 0: successful exit
 < 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(i*100 + j)$,
 if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case $N/P \gg bw$ are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix *A* one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into *P* pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

- 1. Local Phase** : The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space *af*. Mathematically, this is equivalent to reordering the matrix *A* as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of *A* in memory.

2. **Reduced System Phase** : A small ($bw*(P-1)$) system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space *af*. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
3. **Back Substitution Phase**: For a linear system, a local backsubstitution is performed on each processor in parallel.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf .

Syntax

```
call pspttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pdpttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pcpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pzpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

Description

The p?pttrsv routine solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code p?pttrf and is stored in $A(1:n, ja:ja+n-1)$ and *af*. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to *uplo*.

The routine p?pttrf must be called first.

Input Parameters

uplo (global) CHARACTER. Must be 'U' or 'L'.
 If *uplo* = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;
 If *uplo* = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

trans (global) CHARACTER. Must be 'N' or 'C'.
 If *trans* = 'N', solve with $A(1:n, ja:ja+n-1)$;
 If *trans* = 'C' (for complex flavors), solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>d</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv COMPLEX*16 for pzpttrsv. Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size $\geq nb_a$.
<i>e</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv COMPLEX*16 for pzpttrsv. Pointer to the local part of the global vector du storing the upper diagonal of the matrix; must be of size $\geq nb_a$. Globally, $du(n)$ is not referenced, and du must be aligned with d .
<i>ja</i>	(global) INTEGER. The index in the global matrix A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A . If 1D type ($dtype_a = 501$ or 502), then $dlen \geq 7$; If 2D type ($dtype_a = 1$), then $dlen \geq 9$. Contains information on mapping of A to memory. See ScaLAPACK manual for full description and options.
<i>b</i>	(local) REAL for pspttrsv DOUBLE PRECISION for pdpttrsv COMPLEX for pcpttrsv COMPLEX*16 for pzpttrsv. Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.

<i>ib</i>	(global) INTEGER. The row index in the global matrix <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . If 1D type (<i>dtype_b</i> = 502), then <i>dlen</i> ≥ 7; If 2D type (<i>dtype_b</i> = 1), then <i>dlen</i> ≥ 9. Contains information on mapping of <i>B</i> to memory. See ScaLAPACK manual for full description and options.
<i>laf</i>	(local) INTEGER. The size of user-input auxiliary fill-in space <i>af</i> . Must be <i>laf</i> ≥ (<i>nb</i> +2* <i>bw</i>)* <i>bw</i> . If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) REAL for <i>pspttrsv</i> DOUBLE PRECISION for <i>pdpttrsv</i> COMPLEX for <i>pcpttrsv</i> COMPLEX*16 for <i>pzpttrsv</i> . The array <i>work</i> is a temporary workspace array of size <i>lwork</i> . This space may be overwritten in between calls to routines.
<i>lwork</i>	(local or global) INTEGER. The size of the user-input workspace <i>work</i> , must be at least <i>lwork</i> ≥ (10+2*min(100, <i>nrhs</i>))* <i>npcol</i> +4* <i>nrhs</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned.

Output Parameters

<i>d, e</i>	(local). REAL for <i>pspttrsv</i> DOUBLE PRECISION for <i>pdpttrsv</i> COMPLEX for <i>pcpttrsv</i> COMPLEX*16 for <i>pzpttrsv</i> . On exit, these arrays contain information on the factors of the matrix.
<i>af</i>	(local) REAL for <i>pspttrsv</i> DOUBLE PRECISION for <i>pdpttrsv</i> COMPLEX for <i>pcpttrsv</i> COMPLEX*16 for <i>pzpttrsv</i> .

The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization routine `p?pbtrf` and is stored in *af*. If a linear system is to be solved using `p?pttrs` after the factorization routine, *af* must not be altered after the factorization.

<i>b</i>	On exit, this array contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
call pspotf2(uplo, n, a, ia, ja, desca, info)
call pdpotf2(uplo, n, a, ia, ja, desca, info)
call pcpotf2(uplo, n, a, ia, ja, desca, info)
call pzpotf2(uplo, n, a, ia, ja, desca, info)
```

Description

The `p?potf2` routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub (*A*)=*A*(*ia:ia+n-1, ja:ja+n-1*).

The factorization has the form

$\text{sub}(A) = U^*U$, if *uplo* = 'U', or $\text{sub}(A) = L^*L'$, if *uplo* = 'L',

where *U* is an upper triangular matrix, *L* is lower triangular. *X'* denotes transpose (conjugate transpose) of *X*.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix <i>A</i> is stored. = 'U': upper triangle of sub (<i>A</i>) is stored; = 'L': lower triangle of sub (<i>A</i>) is stored.
<i>n</i>	(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed matrix sub (*A*). $n \geq 0$.

a

(local)

REAL for pspotf2

DOUBLE PRECISION for pdpotf2

COMPLEX for pcpotf2

COMPLEX*16 for pzpotf2.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$ containing the local pieces of the *n*-by-*n* symmetric distributed matrix sub(*A*) to be factored.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, ja

(global) INTEGER.

The row and column indices in the global matrix *A* indicating the first row and the first column of the sub(*A*), respectively.

desca

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a

(local)

On exit,

if *uplo* = 'U', the upper triangular part of the distributed matrix contains the Cholesky factor *U*;

if *uplo* = 'L', the lower triangular part of the distributed matrix contains the Cholesky factor *L*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100 + *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?rot

Applies a planar rotation to two distributed vectors.

Syntax

```
call psrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork,
info )
```

```
call pdrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork,
info )
```

Description

p?rot applies a planar rotation defined by *cs* and *sn* to the two distributed vectors sub(*x*) and sub(*y*).

Input Parameters

<i>n</i>	(global) INTEGER The number of elements to operate on when applying the planar rotation to <i>x</i> and <i>y</i> ($n \geq 0$).
<i>x</i>	REAL for psrot DOUBLE PRECISION for pdrot (local) array of size $(j_x - 1) * m_x + i_x + (n - 1) * \text{abs}(inc_x)$ This array contains the entries of the distributed vector sub(<i>x</i>).
<i>ix</i>	(global) INTEGER The global row index of the submatrix of the distributed matrix <i>x</i> to operate on. If $inc_x = 1$, then it is required that $ix = iy$. $1 \leq ix \leq m_x$.
<i>jx</i>	(global) INTEGER The global column index of the submatrix of the distributed matrix <i>x</i> to operate on. If $inc_x = m_x$, then it is required that $jx = jy$. $1 \leq ix \leq n_x$.
<i>descx</i>	(global and local) INTEGER array of size 9 The array descriptor of the distributed matrix <i>x</i> .
<i>incx</i>	(global) INTEGER The global increment for the elements of <i>x</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x . Moreover, it must hold that $inc_x = m_x$ if $inc_y = m_y$ and that $inc_x = 1$ if $inc_y = 1$.
<i>y</i>	REAL for psrot DOUBLE PRECISION for pdrot (local) array of size $(j_y - 1) * m_y + i_y + (n - 1) * \text{abs}(inc_y)$ This array contains the entries of the distributed vector sub(<i>y</i>).
<i>iy</i>	(global) INTEGER

	The global row index of the submatrix of the distributed matrix y to operate on. If $incy = 1$, then it is required that $iy = ix$. $1 \leq iy \leq m_y$.
jy	(global) INTEGER The global column index of the submatrix of the distributed matrix y to operate on. If $incy = m_x$, then it is required that $jy = jx$. $1 \leq jy \leq m_y$.
$descy$	(global and local) INTEGER array of size 9 The array descriptor of the distributed matrix y .
$incy$	(global) INTEGER The global increment for the elements of y . Only two values of $incy$ are supported in this version, namely 1 and m_y . Moreover, it must hold that $incy = m_y$ if $incx = m_x$ and that $incy = 1$ if $incx = 1$.
cs, sn	(global) REAL for psrot DOUBLE PRECISION for pdrot The parameters defining the properties of the planar rotation. It must hold that $0 \leq cs, sn \leq 1$ and that $sn^2 + cs^2 = 1$. The latter is hardly checked in finite precision arithmetics.
$work$	REAL for psrot DOUBLE PRECISION for pdrot (local workspace) array of size $lwork$
$lwork$	(local) INTEGER The length of the workspace array $work$. If $incx = 1$ and $incy = 1$, then $lwork = 2 * m_x$ If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by pxerbla .

OUTPUT Parameters

x	
y	
$work(1)$	On exit, if $info = 0$, $work(1)$ returns the optimal $lwork$
$info$	(global) INTEGER = 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value. If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
call psrscl(n, sa, sx, ix, jx, descx, incx)
call pdrscl(n, sa, sx, ix, jx, descx, incx)
call pcsrscl(n, sa, sx, ix, jx, descx, incx)
call pzdrscl(n, sa, sx, ix, jx, descx, incx)
```

Description

The `p?rscl` routine multiplies an n -element real/complex vector `sub(X)` by the real scalar $1/a$. This is done without overflow or underflow as long as the final result `sub(X)/a` does not overflow or underflow.

`sub(X)` denotes $X(ix:ix+n-1, jx:jx)$, if $incx = 1$,

and $X(ix:ix, jx:jx+n-1)$, if $incx = m_x$.

Input Parameters

<code>n</code>	(global) INTEGER. The number of components of the distributed vector <code>sub(X)</code> . $n \geq 0$.
<code>sa</code>	REAL for <code>psrscl/pcsrscl</code> DOUBLE PRECISION for <code>pdrscl/pzdrscl</code> . The scalar a that is used to divide each component of the vector <code>sub(X)</code> . This parameter must be ≥ 0 .
<code>sx</code>	REAL for <code>psrscl</code> DOUBLE PRECISION for <code>pdrscl</code> COMPLEX for <code>pcsrscl</code> COMPLEX*16 for <code>pzdrscl</code> . Array containing the local pieces of a distributed matrix of size of at least $((jx-1)*m_x + ix + (n-1)*abs(incx))$. This array contains the entries of the distributed vector <code>sub(X)</code> .
<code>ix</code>	(global) INTEGER. The row index of the submatrix of the distributed matrix X to operate on.
<code>jx</code>	(global) INTEGER. The column index of the submatrix of the distributed matrix X to operate on.
<code>descx</code>	(global and local) INTEGER. Array of size 9. The array descriptor for the distributed matrix X .
<code>incx</code>	(global) INTEGER.

The increment for the elements of X . This version supports only two values of $incx$, namely 1 and m_x .

Output Parameters

sx On exit, the result x/a .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).

Syntax

```
call pssygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

Description

The p?sygs2/p?hegs2 routine reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

Here $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$, and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $ibtype = 1$, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$$

and $\text{sub}(A)$ is overwritten by

$\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ - for real flavors, and
 $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ - for complex flavors.

If $ibtype = 2$ or 3 , the problem is

$$\text{sub}(A) * \text{sub}(B) x = \lambda * x \text{ or } \text{sub}(B) * \text{sub}(A) x = \lambda * x$$

and $\text{sub}(A)$ is overwritten by

$U * \text{sub}(A) * U^T$ or $L * T * \text{sub}(A) * L$ - for real flavors and
 $U * \text{sub}(A) * U^H$ or $L * H * \text{sub}(A) * L$ - for complex flavors.

The matrix $\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ (for real flavors), or as $U^H * U$ or $L * L^H$ (for complex flavors) by p?potrf.

Input Parameters

$ibtype$ (global) INTEGER.
 = 1:
 compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ for real subroutines,

and $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ for complex subroutines;

= 2 or 3:

compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$ for real subroutines,

and $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$ for complex subroutines.

uplo

(global) CHARACTER

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored, and how $\text{sub}(B)$ is factorized.

= 'U': Upper triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $U^T * U$ (for real subroutines) or as $U^H * U$ (for complex subroutines).

= 'L': Lower triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $L * L^T$ (for real subroutines) or as $L * L^H$ (for complex subroutines)

n

(global) INTEGER.

The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$. $n \geq 0$.

a

(local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pcheqs2

COMPLEX*16 for pzheqs2.

Pointer into the local memory to an array of size $(lld_a, LOCC(ja+n-1))$.

On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER.

The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

B

(local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pcheqs2

COMPLEX*16 for pzheqs2.

Pointer into the local memory to an array of size $(lld_b, LOCC(jb+n-1))$.

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$ as returned by `p?potrf`.

ib, jb

(global) INTEGER.

The row and column indices in the global matrix B indicating the first row and the first column of the $\text{sub}(B)$, respectively.

descb

(global and local) INTEGER array of size *dlen_*. The array descriptor for the distributed matrix B .

Output Parameters

a

(local)

On exit, if *info* = 0, the transformed matrix is stored in the same format as $\text{sub}(A)$.

info

INTEGER.

= 0: successful exit.

< 0: if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = - (*i**100+ *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
call pssytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

```
call pdsytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

```
call pchetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

```
call pzheta2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Description

The `p?sytd2/p?hetd2` routine reduces a real symmetric/complex Hermitian matrix $\text{sub}(A)$ to symmetric/Hermitian tridiagonal form T by an orthogonal/unitary similarity transformation:

$Q' * \text{sub}(A) * Q = T$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

uplo

(global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored:

	= 'U': upper triangular
	= 'L': lower triangular
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
<i>a</i>	(local) REAL for pssytd2 DOUBLE PRECISION for pdsytd2 COMPLEX for pchetd2 COMPLEX*16 for pzheta2. Pointer into the local memory to an array of size $(lld_a, LOC_c(ja+n-1))$. On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local) REAL for pssytd2 DOUBLE PRECISION for pdsytd2 COMPLEX for pchetd2 COMPLEX*16 for pzheta2. The array <i>work</i> is a temporary workspace array of size <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, if $uplo = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors;
----------	---

if $uplo = 'L'$, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors. See the *Application Notes* below.

d

(local)

REAL for pssytd2/pchetd2

DOUBLE PRECISION for pdsytd2/pzheta2.

Array of size $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix T :

$d(i) = a(i, i)$; d is tied to the distributed matrix A .

e

(local)

REAL for pssytd2/pchetd2

DOUBLE PRECISION for pdsytd2/pzheta2.

Array of size $LOCc(ja+n-1)$,

if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$e(i) = a(i, i+1)$ if $uplo = 'U'$,

$e(i) = a(i+1, i)$ if $uplo = 'L'$.

e is tied to the distributed matrix A .

tau

(local)

REAL for pssytd2

DOUBLE PRECISION for pdsytd2

COMPLEX for pchetd2

COMPLEX*16 for pzheta2.

Array of size $LOCc(ja+n-1)$.

The scalar factors of the elementary reflectors. tau is tied to the distributed matrix A .

$work(1)$

On exit, $work(1)$ returns the minimal and optimal value of $lwork$.

$lwork$

(local or global) INTEGER.

The size of the workspace array $work$.

$lwork$ is local input and must be at least $lwork \geq 3n$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

$info$

(local) INTEGER.

= 0: successful exit

< 0: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(j*100+j)$,

if the i -th argument is a scalar and had an illegal value,

then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of sub (A) on exit are illustrated by the following examples with $n = 5$:

if $uplo='U'$:	if $uplo='L'$:
$\begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix}$	$\begin{bmatrix} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{bmatrix}$

where d and e denotes diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

NOTE

The distributed matrix sub(A) must verify some alignment properties, namely the following expression should be true:

$$(mb_a.eq.nb_a .AND. iroffa.eq.icoffa) \text{ with } iroffa = \text{mod}(ia - 1, mb_a) \text{ and } icoffa = \text{mod}(ja - 1, nb_a).$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trord

Reorders the Schur factorization of a general matrix.

Syntax

```
call pstrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
```

```
call pdtrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
```

Description

p?trord reorders the real Schur factorization of a real matrix $A = Q^*T^*Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

T must be in Schur form (as returned by p?lahqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

This subroutine uses a delay and accumulate procedure for performing the off-diagonal updates.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>compq</i>	(global) CHARACTER*1 = 'V': update the matrix q of Schur vectors; = 'N': do not update q .
<i>select</i>	(global) INTEGER array of size n <i>select</i> specifies the eigenvalues in the selected cluster. To select a real eigenvalue $w(j)$, <i>select</i> (j) must be set to 1. To select a complex conjugate pair of eigenvalues $w(j)$ and $w(j+1)$, corresponding to a 2-by-2 diagonal block, either <i>select</i> (j) or <i>select</i> ($j+1$) or both must be set to 1; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.
<i>para</i>	(global) INTEGER*6 Block parameters: <i>para</i> (1) maximum number of concurrent computational windows allowed in the algorithm; $0 < para(1) \leq \min(nprow, npcol)$ must hold;

<i>para</i> (2)	number of eigenvalues in each window; $0 < para(2) < para(3)$ must hold;
<i>para</i> (3)	window size; $para(2) < para(3) < mb_t$ must hold;
<i>para</i> (4)	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para(4) \leq 100$ must hold;
<i>para</i> (5)	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para(5) \leq mb_t$ must hold.
<i>para</i> (6)	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para(6) \leq para(2)$ must hold.
<i>n</i>	(global) INTEGER The order of the globally distributed matrix <i>t</i> . $n \geq 0$.
<i>t</i>	REAL for pstrord DOUBLE PRECISION for pdtrord (local) array of size $(lld_t, LOC_c(n))$. The local pieces of the global distributed upper quasi-triangular matrix <i>T</i> , in Schur form.
<i>it, jt</i>	(global) INTEGER The row and column index in the global matrix <i>T</i> indicating the first column of <i>T</i> . $it = jt = 1$ must hold (see Application Notes).
<i>desc_t</i>	(global and local) INTEGER array of size <i>dlen_t</i> . The array descriptor for the global distributed matrix <i>T</i> .
<i>q</i>	REAL for pstrord DOUBLE PRECISION for pdtrord (local) array of size $(lld_q, LOC_c(n))$. On entry, if <i>compq</i> = 'V', the local pieces of the global distributed matrix <i>Q</i> of Schur vectors. If <i>compq</i> = 'N', <i>q</i> is not referenced.
<i>iq, jq</i>	(global) INTEGER The column index in the global matrix <i>Q</i> indicating the first column of <i>Q</i> . $iq = jq = 1$ must hold (see Application Notes).
<i>desc_q</i>	(global and local) INTEGER array of size <i>dlen_q</i> .

The array descriptor for the global distributed matrix Q .

work

REAL for `pstrord`
 DOUBLE PRECISION for `pdtord`
 (local workspace) array of size *lwork*

lwork

(local) INTEGER
 The size of the array *work*.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `pxerbla`.

iwork

(local workspace) INTEGER array of size *liwork*

liwork

(local) INTEGER
 The size of the array *iwork*.
 If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by `pxerbla`

OUTPUT Parameters

select

(global) INTEGER array of size *n*
 The (partial) reordering is displayed.

t

On exit, *t* is overwritten by the local pieces of the reordered matrix T , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

q

On exit, if `compq = 'V'`, *q* has been postmultiplied by the global orthogonal transformation matrix which reorders *t*; the leading *m* columns of *q* form an orthonormal basis for the specified invariant subspace.
 If `compq = 'N'`, *q* is not referenced.

wr, wi

REAL for `pstrord`
 DOUBLE PRECISION for `pdtord`
 (global) array of size *n*
 The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix T . The eigenvalues are in principle stored in the same order as on the diagonal of T , with $wr(i) = t(i,i)$ and, if $t(i:i+1,i:i+1)$ is a 2-by-2 diagonal block, $wi(i) > 0$ and $wi(i+1) = -wi(i)$.
 Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

(global) INTEGER
 The size of the specified invariant subspace.

	$0 \leq m \leq n$.
<i>work</i> (1)	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>info</i>	(global) INTEGER = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *1000+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: here we have several possibilities <ul style="list-style-type: none"> • Reordering of <i>t</i> failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); <i>t</i> may have been partially reordered, and <i>wr</i> and <i>wi</i> contain the eigenvalues in the same order as in <i>t</i>. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}. • A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}. • If <i>info</i> = <i>n</i>+1, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $mb_t = nb_t = mb_q = nb_q$
- $rsrc_t = rsrc_q$
- $csrc_t = csrc_q$

All matrices must be blocked by a block factor larger than or equal to two (3). This is to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of *t* and *q*, i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since [p?lahqr](#) does not compute Schur forms of submatrices anyway.

Parallel execution recommendations:

- Use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.
- In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

```
call pstrsen( job, compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr,
wi, m, s, sep, work, lwork, iwork, liwork, info )
```

```
call pdtrsen( job, compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr,
wi, m, s, sep, work, lwork, iwork, liwork, info )
```

Description

`p?trsen` reorders the real Schur factorization of a real matrix $A = Q^*T^*Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace. The reordering is performed by `p?trord`.

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur form (as returned by `p?lahqr`), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

job (global) CHARACTER*1
 Specifies whether condition numbers are required for the cluster of eigenvalues (*s*) or the invariant subspace (*sep*):
 = 'N': no condition numbers are required;
 = 'E': only the condition number for the cluster of eigenvalues is computed (*s*);
 = 'V': only the condition number for the invariant subspace is computed (*sep*);
 = 'B': condition numbers for both the cluster and the invariant subspace are computed (*s* and *sep*).

compq (global) CHARACTER*1
 = 'V': update the matrix *q* of Schur vectors;
 = 'N': do not update *q*.

select (global) LOGICAL array of size *n*

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue $w(j)$, *select(j)* must be set to `.TRUE.`. To select a complex conjugate pair of eigenvalues $w(j)$ and $w(j+1)$, corresponding to a 2-by-2 diagonal block, either *select(j)* or *select(j+1)* or both must be set to `.TRUE.`; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para

(global) INTEGER*6

Block parameters:

para(1) maximum number of concurrent computational windows allowed in the algorithm; $0 < para(1) \leq \min(NPROW, NPCOL)$ must hold;

para(2) number of eigenvalues in each window; $0 < para(2) < para(3)$ must hold;

para(3) window size; $para(2) < para(3) < mb_t$ must hold;

para(4) minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para(4) \leq 100$ must hold;

para(5) width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para(5) \leq mb_t$ must hold.

para(6) the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size $0 < para(6) \leq para(2)$ must hold;

n

(global) INTEGER

The order of the globally distributed matrix t . $n \geq 0$.

t

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local) array of size $(lld_t, LOC_c(n))$.

The local pieces of the global distributed upper quasi-triangular matrix T , in Schur form.

it, jt

(global) INTEGER

The row and column index in the global matrix T indicating the first column of T . $it = jt = 1$ must hold (see Application Notes).

desct

(global and local) INTEGER array of size $dlen_$.

The array descriptor for the global distributed matrix T .

q

REAL for pstrsen

DOUBLE PRECISION for `pdtrsen`

(local) array of size $(lld_q, LOC_c(n))$.

On entry, if `compq = 'V'`, the local pieces of the global distributed matrix Q of Schur vectors.

If `compq = 'N'`, q is not referenced.

`iq, jq`

(global) INTEGER

The column index in the global matrix Q indicating the first column of Q . $iq = jq = 1$ must hold (see Application Notes).

`descq`

(global and local) INTEGER array of size `dlen_`.

The array descriptor for the global distributed matrix Q .

`work`

REAL for `pstrsen`

DOUBLE PRECISION for `pdtrsen`

(local workspace) array of size `lwork`

`lwork`

(local) INTEGER

The size of the array `work`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `pxerbla`.

`iwork`

(local workspace) INTEGER array of size `liwork`

`liwork`

(local) INTEGER

The size of the array `iwork`.

If `liwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `iwork` array, returns this value as the first entry of the `iwork` array, and no error message related to `liwork` is issued by `pxerbla`.

OUTPUT Parameters

`t`

`t` is overwritten by the local pieces of the reordered matrix T , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

`q`

On exit, if `compq = 'V'`, q has been postmultiplied by the global orthogonal transformation matrix which reorders t ; the leading m columns of q form an orthonormal basis for the specified invariant subspace.

If `compq = 'N'`, q is not referenced.

`wr, wi`

REAL for `pstrsen`

DOUBLE PRECISION for `pdtrsen`

(global) array of size `n`

The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix T . The eigenvalues are in principle stored in the same order as on the diagonal of T , with $w_r(i) = t(i,i)$ and, if $t(i:i+1,i:i+1)$ is a 2-by-2 diagonal block, $w_i(i) > 0$ and $w_i(i+1) = -w_i(i)$.

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

<i>m</i>	(global) INTEGER The size of the specified invariant subspace. $0 \leq m \leq n$.
<i>s</i>	REAL for <i>pstrsen</i> DOUBLE PRECISION for <i>pdtrsen</i> (global) If <i>job</i> = 'E' or 'B', <i>s</i> is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. <i>s</i> cannot underestimate the true reciprocal condition number by more than a factor of \sqrt{n} . If $m = 0$ or n , $s = 1$. If <i>job</i> = 'N' or 'V', <i>s</i> is not referenced.
<i>sep</i>	REAL for <i>pstrsen</i> DOUBLE PRECISION for <i>pdtrsen</i> (global) If <i>job</i> = 'V' or 'B', <i>sep</i> is the estimated reciprocal condition number of the specified invariant subspace. If $m = 0$ or n , $sep = \text{norm}(t)$. If <i>job</i> = 'N' or 'E', <i>sep</i> is not referenced.
<i>work</i> (1)	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal <i>lwork</i> .
<i>iwork</i> (1)	On exit, if <i>info</i> = 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>info</i>	(global) INTEGER = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then $info = -(j*1000+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then $info = -i$. > 0: here we have several possibilities <ul style="list-style-type: none"> • Reordering of <i>t</i> failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); <i>t</i> may have been partially reordered, and <i>w_r</i> and <i>w_i</i> contain the eigenvalues in the same order as in <i>t</i>. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}. • A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}.

- If $info = n+1$, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $mb_t = nb_t = mb_q = nb_q$
- $rsrc_t = rsrc_q$
- $csrc_t = csrc_q$

All matrices must be blocked by a block factor larger than or equal to two (3). This to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of t and q , i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since `p?lahqr` does not compute Schur forms of submatrices anyway.

For parallel execution, use a square grid, if possible, for maximum performance. The block parameters in `para` should be kept well below the data distribution block size.

In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
call pstrti2(uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2(uplo, diag, n, a, ia, ja, desca, info)
call pctrti2(uplo, diag, n, a, ia, ja, desca, info)
call pztrti2(uplo, diag, n, a, ia, ja, desca, info)
```

Description

The `p?trti2` routine computes the inverse of a real/complex upper or lower triangular block matrix sub (A) = $A(ia:ia+n-1, ja:ja+n-1)$.

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Specifies whether the matrix sub (A) is upper or lower triangular. = 'U': sub (A) is upper triangular = 'L': sub (A) is lower triangular.
<code>diag</code>	(global) CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': sub (A) is non-unit triangular

	= 'U': sub (A) is unit triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, i.e., the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<i>a</i>	(local) REAL for pstrti2 DOUBLE PRECISION for pdtrti2 COMPLEX for pctrti2 COMPLEX*16 for pztrti2. Pointer into the local memory to an array, size $(lld_a, LOCC(ja+n-1))$. On entry, this array contains the local pieces of the triangular matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of the matrix $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of the matrix $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced. If <i>diag</i> = 'U', the diagonal elements of $\text{sub}(A)$ are not referenced either and are assumed to be 1.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	On exit, the (triangular) inverse of the original matrix, in the same storage format.
<i>info</i>	INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then $info = -(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
call slamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
```

```
call dlamsh(s, lds, nbulge, jblk, h, ldh, n, ulp)
```

Description

The ?lamsh routine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

<i>s</i>	(local) REAL for slamsh DOUBLE PRECISION for dlamsh Array of size (<i>lds</i> , *). On entry, the matrix of shifts. Only the 2x2 diagonal of <i>s</i> is referenced. It is assumed that <i>s</i> has <i>jblk</i> double shifts (size 2).
<i>lds</i>	(local) INTEGER. On entry, the leading dimension of <i>S</i> ; unchanged on exit. $1 < \text{nbulge} \leq \text{jblk} \leq \text{lds}/2$.
<i>nbulge</i>	(local) INTEGER. On entry, the number of bulges to send through <i>h</i> (>1). <i>nbulge</i> should be less than the maximum determined (<i>jblk</i>). $1 < \text{nbulge} \leq \text{jblk} \leq \text{lds}/2$.
<i>jblk</i>	(local) INTEGER. On entry, the leading dimension of <i>S</i> ; unchanged on exit.
<i>h</i>	(local) REAL for slamsh DOUBLE PRECISION for dlamsh Array of size <i>lds</i> by <i>n</i> . On entry, the local matrix to apply the shifts on. <i>h</i> should be aligned so that the starting row is 2.
<i>ldh</i>	(local) INTEGER. On entry, the leading dimension of <i>H</i> ; unchanged on exit.

<i>n</i>	(local) INTEGER. On entry, the size of <i>H</i> . If all the bulges are expected to go through, <i>n</i> should be at least $4n_{bulge}+2$. Otherwise, <i>nbulge</i> may be reduced by this routine.
<i>ulp</i>	(local) REAL for <i>slamsh</i> DOUBLE PRECISION for <i>dlamsh</i> On entry, machine precision. Unchanged on exit.

Output Parameters

<i>s</i>	On exit, the data is rearranged in the best order for applying.
<i>nbulge</i>	On exit, the maximum number of bulges that can be sent through.
<i>h</i>	On exit, the data is destroyed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?laqr6

Performs a single small-bulge multi-shift QR sweep collecting the transformations.

Syntax

```
call slaqr6( job, wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

```
call dlaqr6( job, wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Description

This auxiliary subroutine called by [p?laqr5](#) performs a single small-bulge multi-shift QR sweep, moving the chain of bulges from top to bottom in the submatrix $H(k_{top}:k_{bot},k_{top}:k_{bot})$, collecting the transformations in the matrix *V* or accumulating the transformations in the matrix *Z* (see below).

This is a modified version of [?laqr5](#) from LAPACK 3.1.

Input Parameters

<i>job</i>	CHARACTER scalar Set the kind of job to do in ?laqr6 , as follows: <i>job</i> = 'I': Introduce and chase bulges in submatrix <i>job</i> = 'C': Chase bulges from top to bottom of submatrix <i>job</i> = 'O': Chase bulges off submatrix
<i>wantt</i>	LOGICAL scalar

	<p><i>wantt</i> = .TRUE. if the quasi-triangular Schur factor is being computed. <i>wantt</i> is set to .FALSE. otherwise.</p>
<i>wantz</i>	<p>LOGICAL scalar</p> <p><i>wantz</i> = .TRUE. if the orthogonal Schur factor is being computed. <i>wantz</i> is set to .FALSE. otherwise.</p>
<i>kacc22</i>	<p>INTEGER with value 0, 1, or 2.</p> <p>Specifies the computation mode of far-from-diagonal orthogonal updates.</p> <p>= 0: ?laqr6 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.</p> <p>= 1: ?laqr6 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.</p> <p>= 2: ?laqr6 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.</p>
<i>n</i>	<p>INTEGER scalar</p> <p><i>n</i> is the order of the Hessenberg matrix <i>H</i> upon which this subroutine operates.</p>
<i>ktop, kbot</i>	<p>INTEGER scalar</p> <p>These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either <i>ktop</i> = 1 or $H(ktop,ktop-1) = 0$ and either <i>kbot</i> = <i>n</i> or $H(kbot+1,kbot) = 0$.</p>
<i>nshfts</i>	<p>INTEGER scalar</p> <p><i>nshfts</i> gives the number of simultaneous shifts. <i>nshfts</i> must be positive and even.</p>
<i>sr, si</i>	<p>REAL for slaqr6</p> <p>DOUBLE PRECISION for dlaqr6</p> <p>Array of size <i>nshfts</i></p> <p><i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.</p>
<i>h</i>	<p>REAL for slaqr6</p> <p>DOUBLE PRECISION for dlaqr6</p> <p>Array of size (<i>ldh</i>,<i>n</i>)</p> <p>On input <i>h</i> contains a Hessenberg matrix .</p>
<i>ldh</i>	<p>INTEGER scalar</p> <p><i>ldh</i> is the leading dimension of <i>H</i> just as declared in the calling procedure. <i>ldh</i> ≥ max(1,<i>n</i>).</p>
<i>iloz, ihiz</i>	<p>INTEGER scalar</p>

Specify the rows of z to which transformations must be applied if $wantzis$ `.TRUE.`.. $1 \leq iloz \leq ihiz \leq n$

z	<p>REAL for <code>slaqr6</code></p> <p>DOUBLE PRECISION for <code>dlaqr6</code></p> <p>Array of size $(ldz,ktop)$</p> <p>If $wantz=$ <code>.TRUE.</code>, then the QR sweep orthogonal similarity transformation is accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$, stored in the array z, from the right.</p> <p>If $wantz=$ <code>.FALSE.</code>, then z is unreferenced.</p>
ldz	<p>INTEGER scalar</p> <p>ldz is the leading dimension of z just as declared in the calling procedure. $ldz \geq n$.</p>
v	<p>REAL for <code>slaqr6</code></p> <p>DOUBLE PRECISION for <code>dlaqr6</code></p> <p>(workspace) array of size $(ldv,nshfts/2)$</p>
ldv	<p>INTEGER scalar</p> <p>ldv is the leading dimension of v as declared in the calling procedure. $ldv \geq 3$.</p>
u	<p>REAL for <code>slaqr6</code></p> <p>DOUBLE PRECISION for <code>dlaqr6</code></p> <p>(workspace) array of size $(ldu,3*nshfts-3)$</p>
ldu	<p>INTEGER scalar</p> <p>ldu is the leading dimension of u just as declared in the calling subroutine. $ldu \geq 3*nshfts-3$.</p>
nh	<p>INTEGER scalar</p> <p>nh is the number of columns in array wh available for workspace. $nh \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.</p>
wh	<p>REAL for <code>slaqr6</code></p> <p>DOUBLE PRECISION for <code>dlaqr6</code></p> <p>(workspace) array of size $(ldwh,nh)$</p>
$ldwh$	<p>INTEGER scalar</p> <p>Leading dimension of wh just as declared in the calling procedure. $ldwh \geq 3*nshfts-3$.</p>
nv	<p>INTEGER scalar</p>

nv is the number of rows in wv available for workspace. $nv \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

wv REAL for `slaqr6`
 DOUBLE PRECISION for `dlaqr6`
 (workspace) array of size $(ldwv, 3 * nshfts - 3)$

$ldwv$ INTEGER scalar
 $ldwv$ is the leading dimension of wv as declared in the in the calling subroutine. $ldwv \geq nv$.

OUTPUT Parameters

h A multi-shift QR sweep with shifts $sr(j) + i * si(j)$ is applied to the isolated diagonal block in rows and columns $ktop$ through $kbot$.

z If `wantzis .TRUE.`, then the QR sweep orthogonal/unitary similarity transformation is accumulated into the matrix $Z(iloz:ihiz, kbot:ktop)$ from the right.
 If `wantzis .FALSE.`, then z is unreferenced.

Application Notes

Notes

Based on contributions by Karen Braman and Ralph Byers, Department of Mathematics, University of Kansas, USA Robert Granat, Department of Computing Science and HPC2N, Umea University, Sweden

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lar1va

Computes scaled eigenvector corresponding to given eigenvalue.

Syntax

```
call slar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

```
call dlar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcrr, work )
```

Description

?slar1va computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $LDL^T - \lambda I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, r corresponds to the index where the eigenvector is largest in magnitude. The following steps accomplish this computation :

1. Stationary qd transform, $LDL^T - \lambda I = L_+ D_+ L_+^T$,
2. Progressive qd transform, $LDL^T - \lambda I = U \cdot D \cdot U^T$,
3. Computation of the diagonal elements of the inverse of $LDL^T - \lambda I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.

4. Computation of the (scaled) r -th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n	INTEGER	The order of the matrix LDL^T .
$b1$	INTEGER	First index of the submatrix of LDL^T .
bn	INTEGER	Last index of the submatrix of LDL^T .
$lambda$	REAL for slarlva DOUBLE PRECISION for dlarlva	The shift λ . In order to compute an accurate eigenvector, $lambda$ should be a good approximation to an eigenvalue of LDL^T .
l	REAL for slarlva DOUBLE PRECISION for dlarlva	Array of size $n-1$ The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L , in elements 1 to $n-1$.
d	REAL for slarlva DOUBLE PRECISION for dlarlva	Array of size n The n diagonal elements of the diagonal matrix D .
ld	REAL for slarlva DOUBLE PRECISION for dlarlva	Array of size $n-1$ The $n-1$ elements $l(i)*d(i)$.
lld	REAL for slarlva DOUBLE PRECISION for dlarlva	Array of size $n-1$ The $n-1$ elements $l(i)*l(i)*d(i)$.
$pivmin$	REAL for slarlva DOUBLE PRECISION for dlarlva	The minimum pivot in the Sturm sequence.
$gaptol$	REAL for slarlva DOUBLE PRECISION for dlarlva	

Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z REAL for `slar1va`
 DOUBLE PRECISION for `dlar1va`
 Array of size *n*
 On input, all entries of *z* must be set to 0.

wantnc LOGICAL
 Specifies whether *negcnt* has to be computed.

r INTEGER
 The twist index for the twisted factorization used to compute *z*.
 On input, $0 \leq r \leq n$. If *r* is input as 0, *r* is set to the index where $(LDL^T - \sigma I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, *r* is unchanged.
 Ideally, *r* designates the position of the maximum entry in the eigenvector.

work REAL for `slar1va`
 DOUBLE PRECISION for `dlar1va`
 (Workspace) array of size $4*n$

OUTPUT Parameters

z On output, *z* contains the (scaled) *r*-th column of the inverse. The scaling is such that $z(r)$ equals 1.

negcnt INTEGER
 If *wantncis* `.TRUE.` then *negcnt* = the number of pivots $< pivmin$ in the matrix factorization LDL^T , and *negcnt* = -1 otherwise.

ztz REAL for `slar1va`
 DOUBLE PRECISION for `dlar1va`
 The square of the 2-norm of *z*.

mingma REAL for `slar1va`
 DOUBLE PRECISION for `dlar1va`
 The reciprocal of the largest (in magnitude) diagonal element of the inverse of $LDL^T - \sigma I$.

r On output, *r* contains the twist index used to compute *z*.

isuppz INTEGER array of size 2
 The support of the vector in *z*, i.e., the vector *z* is non-zero only in elements *isuppz*(1) through *isuppz*(2).

nrminv REAL for `slar1va`
 DOUBLE PRECISION for `dlar1va`

nrminv = 1/SQRT(*ztz*)

resid REAL for *slar1va*
DOUBLE PRECISION for *dlar1va*
The residual of the FP vector.
 $resid = ABS(mingma)/SQRT(ztz)$

rqcorr REAL for *slar1va*
DOUBLE PRECISION for *dlar1va*
The Rayleigh Quotient correction to *lambda*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?laref

Applies Householder reflectors to matrices on either their rows or columns.

Syntax

call *slaref*(*type*, *a*, *lda*, *wantz*, *z*, *ldz*, *block*, *irow1*, *icoll*, *istart*, *istop*, *itmp1*, *itmp2*, *liloz*, *lihiz*, *vecs*, *v2*, *v3*, *t1*, *t2*, *t3*)

call *dlaref*(*type*, *a*, *lda*, *wantz*, *z*, *ldz*, *block*, *irow1*, *icoll*, *istart*, *istop*, *itmp1*, *itmp2*, *liloz*, *lihiz*, *vecs*, *v2*, *v3*, *t1*, *t2*, *t3*)

Description

The *?laref* routine applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type (global) CHARACTER*1.
If *type* = 'R', apply reflectors to the rows of the matrix (apply from left). Otherwise, apply reflectors to the columns of the matrix. Unchanged on exit.

a (global) REAL for *slaref*
DOUBLE PRECISION for *dlaref*
Array of size (*lda*, *).
On entry, the matrix to receive the reflections.

lda (local) INTEGER.
On entry, the leading dimension of *A*; unchanged on exit.

wantz (global) LOGICAL.
If *wantz* = .TRUE., apply any column reflections to *Z* as well.
If *wantz* = .FALSE., do no additional work on *Z*.

<i>z</i>	<p>(global) REAL for <i>slaref</i> DOUBLE PRECISION for <i>dlaref</i></p> <p>Array of size (<i>ldz</i>, *).</p> <p>Referenced only if <i>type</i> differs from 'R'. On entry, the second matrix to receive column reflections.</p>
<i>ldz</i>	<p>(local) INTEGER.</p> <p>On entry, the leading dimension of <i>Z</i>; unchanged on exit.</p>
<i>block</i>	<p>(global) LOGICAL.</p> <p>= .TRUE. : apply several reflectors at once and read their data from the <i>vecs</i> array;</p> <p>= .FALSE. : apply the single reflector given by <i>v2</i>, <i>v3</i>, <i>t1</i>, <i>t2</i>, and <i>t3</i>.</p>
<i>irow1</i>	<p>(local) INTEGER.</p> <p>On entry, the local row element of the matrix <i>A</i>.</p>
<i>icoll</i>	<p>(local) INTEGER.</p> <p>On entry, the local column element of the matrix <i>A</i>.</p>
<i>istart</i>	<p>(global) INTEGER.</p> <p>Specifies the "number" of the first reflector.</p> <p><i>istart</i> is used as an index into <i>vecs</i> if <i>block</i> is set. <i>istart</i> is ignored if <i>blockis</i> .FALSE..</p>
<i>istop</i>	<p>(global) INTEGER.</p> <p>Specifies the "number" of the last reflector.</p> <p><i>istop</i> is used as an index into <i>vecs</i> if <i>block</i> is set. <i>istop</i> is ignored if <i>blockis</i> .FALSE..</p>
<i>itmp1</i>	<p>(local) INTEGER.</p> <p>Starting range into <i>A</i>. For rows, this is the local first column. For columns, this is the local first row.</p>
<i>itmp2</i>	<p>(local) INTEGER.</p> <p>Ending range into <i>A</i>. For rows, this is the local last column. For columns, this is the local last row.</p>
<i>liloz</i> , <i>lihiz</i>	<p>(local). INTEGER.</p> <p>Serve the same purpose as <i>itmp1</i>, <i>itmp2</i> but for <i>Z</i> when <i>wantz</i> is set.</p>
<i>vecs</i>	<p>(global)</p> <p>REAL for <i>slaref</i> DOUBLE PRECISION for <i>dlaref</i>.</p> <p>Array of size 3 * <i>n</i> (matrix size). This array holds the size 3 reflectors one after another and is only accessed when <i>block</i> is .TRUE..</p>

`v2,v3,t1,t2,t3` (global)
 REAL for `slaref`
 DOUBLE PRECISION for `dlaref`.
 These parameters hold information on a single size 3 Householder reflector and are read when `blockis` `.FALSE.`, and overwritten when `block` is `.TRUE.`.

Output Parameters

`a` On exit, the updated matrix.
`z` Changed only if `wantz` is set. If `wantz` is `.FALSE.`, `z` is not referenced.
`irow1` Undefined.
`icoll` Undefined.
`v2,v3,t1,t2,t3` These parameters are read when `blockis` `.FALSE.`, and overwritten when `block` is `.TRUE.`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrb2

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
            iwork, pivmin, lgpvmn, lgspdm, twist, info )
```

```
call dlarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,
            iwork, pivmin, lgpvmn, lgspdm, twist, info )
```

Description

Given the relatively robust representation (RRR) LDL^T , `?larrb2` does "limited" bisection to refine the eigenvalues of LDL^T , $w(ifirst - offset)$ through $w(ilast - offset)$, to more accuracy. Initial guesses for these eigenvalues are input in `w`, the corresponding estimate of the error in these guesses and their gaps are input in `werr` and `wgap`, respectively. During bisection, intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays `w` and `werr` respectively.

NOTE

There are very few minor differences between `larrb` from LAPACK and this current subroutine `?larrb2`. The most important reason for creating this nearly identical copy is profiling: in the ScaLAPACK MRRR algorithm, eigenvalue computation using `?larrb2` is used for refinement in the construction of the representation tree, as opposed to the initial computation of the eigenvalues for the root RRR which uses `?larrb`. When profiling, this allows an easy quantification of refinement work vs. computing eigenvalues of the root.

Input Parameters

<i>n</i>	INTEGER The order of the matrix.
<i>d</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array of size <i>n</i> . The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>lld</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array of size <i>n</i> -1. The (<i>n</i> -1) elements $l_i^*l_i^*d(i)$.
<i>ifirst</i>	INTEGER The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER The index of the last eigenvalue to be computed.
<i>rtol1, rtol2</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Tolerance for the convergence of the bisection intervals. An interval [<i>left</i> , <i>right</i>] has converged if $right - left < \max(rtol1 * gap, rtol2 * \max(left , right))$ where <i>gap</i> is the (estimated) distance to the nearest eigenvalue.
<i>offset</i>	INTEGER Offset for the arrays <i>w</i> , <i>wgap</i> and <i>werr</i> , i.e., the <i>ifirst</i> - <i>offset</i> through <i>ilast</i> - <i>offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array of size <i>n</i> On input, $w(ifirst - offset)$ through $w(ilast - offset)$ are estimates of the eigenvalues of LDL^T indexed <i>ifirst</i> through <i>ilast</i> .
<i>wgap</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array of size <i>n</i> -1. On input, the (estimated) gaps between consecutive eigenvalues of LDL^T , i.e., $wgap(I - offset)$ is the gap between eigenvalues <i>I</i> and <i>I</i> + 1. Note that if <i>ifirst</i> = <i>ilast</i> then $wgap(ifirst - offset)$ must be set to zero.
<i>werr</i>	REAL for slarrb2

	DOUBLE PRECISION for dlarrb2
	Array of size n .
	On input, $werr(ifirst - offset)$ through $werr(ilast - offset)$ are the errors in the estimates of the corresponding elements in w .
<i>work</i>	REAL for slarrb2
	DOUBLE PRECISION for dlarrb2
	(workspace) array of size $4*n$.
	Workspace.
<i>iwork</i>	(workspace) INTEGER array of size $2*n$.
	Workspace.
<i>pivmin</i>	REAL for slarrb2
	DOUBLE PRECISION for dlarrb2
	The minimum pivot in the Sturm sequence.
<i>lgpvmn</i>	REAL for slarrb2
	DOUBLE PRECISION for dlarrb2
	Logarithm of <i>pivmin</i> , precomputed.
<i>lgspdm</i>	REAL for slarrb2
	DOUBLE PRECISION for dlarrb2
	Logarithm of the spectral diameter, precomputed.
<i>twist</i>	INTEGER
	The twist index for the twisted factorization that is used for the <i>negcount</i> .
	$twist = n$: Compute <i>negcount</i> from $LDL^T - \lambda I = L_+ D_+ L_+^T$
	$twist = 1$: Compute <i>negcount</i> from $LDL^T - \lambda I = U \cdot D \cdot U^T$
	$twist = r, 1 < r < n$: Compute <i>negcount</i> from $LDL^T - \lambda I = N_r A_r N_r^T$

OUTPUT Parameters

<i>w</i>	On output, the eigenvalue estimates in <i>w</i> are refined.
<i>wgap</i>	On output, the eigenvalue gaps in <i>wgap</i> are refined.
<i>werr</i>	On output, the errors in <i>werr</i> are refined.
<i>info</i>	INTEGER
	Error flag.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrrd2

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit,
             isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
```

```
call dlarrrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplit,
             isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
```

Description

?larrrd2 computes the eigenvalues of a symmetric tridiagonal matrix T to limited initial accuracy. This is an auxiliary code to be called from [larre2a](#).

?larrrd2 has been created using the LAPACK code [larrrd](#) which itself stems from [stebz](#). The motivation for creating ?larrrd2 is efficiency: When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, ?larrrd2 can skip over blocks which contain none of the eigenvalues from DOL to DOU for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size like 2x2), the gain can be substantial.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>range</i>	<p>CHARACTER</p> <p>= 'A': ("All") all eigenvalues will be found.</p> <p>= 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.</p> <p>= 'I': ("Index") the il-th through iu-th eigenvalues (of the entire matrix) will be found.</p>
<i>order</i>	<p>CHARACTER</p> <p>= 'B': ("By Block") the eigenvalues will be grouped by split-off block (see <i>iblock</i>, <i>isplit</i>) and ordered from smallest to largest within the block.</p> <p>= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.</p>
<i>n</i>	<p>INTEGER</p> <p>The order of the tridiagonal matrix T. $n \geq 0$.</p>

<i>vl, vu</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>If <i>range</i>='V', the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to <i>vl</i>, or greater than <i>vu</i>, will not be returned. $vl < vu$.</p> <p>Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il, iu</i>	<p>INTEGER</p> <p>If <i>range</i>='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.</p> <p>$1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$ if $n = 0$.</p> <p>Not referenced if <i>range</i> = 'A' or 'V'.</p>
<i>gers</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size $2*n$</p> <p>The n Gerschgorin intervals (the i-th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).</p>
<i>reltol</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least $\text{radix} * \text{machine epsilon}$.</p>
<i>d</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size n</p> <p>The n diagonal elements of the tridiagonal matrix T.</p>
<i>e</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size $n-1$</p> <p>The $(n-1)$ off-diagonal elements of the tridiagonal matrix T.</p>
<i>e2</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size $n-1$</p> <p>The $(n-1)$ squared off-diagonal elements of the tridiagonal matrix T.</p>
<i>pivmin</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>The minimum pivot allowed in the sturm sequence for T.</p>

<i>nsplit</i>	<p>INTEGER</p> <p>The number of diagonal blocks in the matrix T.</p> <p>$1 \leq nsplit \leq n$.</p>
<i>isplit</i>	<p>INTEGER Array of size n</p> <p>The splitting points, at which T breaks up into submatrices.</p> <p>The first submatrix consists of rows/columns 1 to <i>isplit</i>(1), the second of rows/columns <i>isplit</i>(1)+1 through <i>isplit</i>(2), etc., and the <i>nsplit</i>-th submatrix consists of rows/columns <i>isplit</i>(<i>nsplit</i>-1)+1 through <i>isplit</i>(<i>nsplit</i>)=n.</p> <p>(Only the first <i>nsplit</i> elements will actually be used, but since the user cannot know a priori what value <i>nsplit</i> will have, n words must be reserved for <i>isplit</i>.)</p>
<i>work</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>(workspace) Array of size $4*n$</p>
<i>iwork</i>	<p>(workspace) INTEGER Array of size $3*n$</p>
<i>dol, dou</i>	<p>INTEGER</p> <p>Specifying an index range <i>dol:dou</i> allows the user to work on only a selected part of the representation tree.</p> <p>Otherwise, the setting <i>dol</i>=1, <i>dou</i>=n should be applied.</p> <p>Note that <i>dol</i> and <i>dou</i> refer to the order in which the eigenvalues are stored in W.</p>

OUTPUT Parameters

<i>m</i>	<p>INTEGER</p> <p>The actual number of eigenvalues found. $0 \leq m \leq n$.</p> <p>(See also the description of <i>info</i>=2,3.)</p>
<i>w</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size n</p> <p>On exit, the first m elements of <i>w</i> will contain the eigenvalue approximations. ?larrd2 computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j. The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j)/2$. The corresponding error is bounded by $werr(j) = \text{abs}(a_j - b_j)/2$.</p>
<i>werr</i>	<p>REAL for slarrd2</p> <p>DOUBLE PRECISION for dlarrd2</p> <p>Array of size n</p>

	The error bound on the corresponding eigenvalue approximation in w .
wl, wu	REAL for slarrd2 DOUBLE PRECISION for dlarrd2 The interval $(wl, wu]$ contains all the wanted eigenvalues. If $range='V'$, then $wl=vl$ and $wu=vu$. If $range='A'$, then wl and wu are the global Gerschgorin bounds on the spectrum. If $range='I'$, then wl and wu are computed by SLAEBZ from the index range specified.
$iblock$	INTEGER Array of size n At each row/column j where $e(j)$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit, if $info = 0$, $iblock(i)$ specifies to which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs. ($?larrd2$ may use the remaining $n-m$ elements as workspace.)
$indexw$	INTEGER Array of size n The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=j$ and $iblock(i)=k$ imply that the i -th eigenvalue $w(i)$ is the j -th eigenvalue in block k .
$info$	INTEGER = 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value > 0: some or all of the eigenvalues failed to converge or were not computed: <ul style="list-style-type: none"> • = 1 or 3: Bisection failed to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances. • = 2 or 3: $range='I'$ only: Not all of the eigenvalues $il:iu$ were found. • = 4: $range='I'$, and the Gershgorin interval initially used was too small. No eigenvalues were computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larre2

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```
call slarre2( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
            isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

call `dlarre2(range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info)`

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , `?larre2` sets, via `?larra`, "small" off-diagonal elements to zero. For each block T_i , it finds

- a suitable shift at one end of the block's spectrum,
- the root RRR, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then returned to `?stegr2` to compute the eigenvectors T .

`?larre2` is more suitable for parallel computation than the original LAPACK code for computing the root RRR and its eigenvalues. When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, `?larre2` can skip over blocks which contain none of the eigenvalues from `dol` to `dou` for which the processor is responsible. In extreme cases (such as large matrices consisting of many blocks of small size, e.g. 2x2), the gain can be substantial.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>range</i>	<p>CHARACTER</p> <p>= 'A': ("All") all eigenvalues will be found.</p> <p>= 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.</p> <p>= 'I': ("Index") the <i>il</i>-th through <i>iu</i>-th eigenvalues (of the entire matrix) will be found.</p>
<i>n</i>	<p>INTEGER</p> <p>The order of the matrix. $n > 0$.</p>
<i>vl, vu</i>	<p>REAL for <code>slarre2</code></p> <p>DOUBLE PRECISION for <code>dlarre2</code></p> <p>If <i>range</i>='V', the lower and upper bounds for the eigenvalues.</p> <p>Eigenvalues less than or equal to <i>vl</i>, or greater than <i>vu</i>, will not be returned. $vl < vu$.</p>
<i>il, iu</i>	<p>INTEGER</p> <p>If <i>range</i>='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.</p>

	$1 \leq i_l \leq i_u \leq n.$
<i>d</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>Array of size <i>n</i></p> <p>The <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>.</p>
<i>e</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>Array of size <i>n</i></p> <p>The first (<i>n</i>-1) entries contain the subdiagonal elements of the tridiagonal matrix <i>T</i>; <i>e</i>(<i>n</i>) need not be set.</p>
<i>e2</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>Array of size <i>n</i></p> <p>The first (<i>n</i>-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix <i>T</i>; <i>e2</i>(<i>n</i>) need not be set.</p>
<i>rtol1, rtol2</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>Parameters for bisection.</p> <p>An interval [<i>left, right</i>] has converged if $right-left < \max(rtol1*gap, rtol2*\max(left , right))$</p>
<i>spltol</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>The threshold for splitting.</p>
<i>dol, dou</i>	<p>INTEGER</p> <p>Specifying an index range <i>dol:dou</i> allows the user to work on only a selected part of the representation tree. Otherwise, the setting <i>dol</i>=1, <i>dou</i>=<i>n</i> should be applied.</p> <p>Note that <i>dol</i> and <i>dou</i> refer to the order in which the eigenvalues are stored in <i>w</i>.</p>
<i>work</i>	<p>REAL for slarre2</p> <p>DOUBLE PRECISION for dlarre2</p> <p>Workspace array of size $6*n$</p>
<i>iwork</i>	<p>INTEGER</p> <p>Workspace array of size $5*n$</p>

OUTPUT Parameters

<i>vl, vu</i>	If <i>range</i> ='I' or ='A', <i>?larre2</i> contains bounds on the desired part of the spectrum.
<i>d</i>	The <i>n</i> diagonal elements of the diagonal matrices D_j .
<i>e</i>	<i>e</i> contains the subdiagonal elements of the unit bidiagonal matrices L_j . The entries $e(\textit{isplit}(i))$, $1 \leq i \leq \textit{nsplit}$, contain the base points σ_i on output.
<i>e2</i>	The entries $e2(\textit{isplit}(i))$, $1 \leq i \leq \textit{nsplit}$, are set to zero.
<i>nsplit</i>	INTEGER The number of blocks <i>T</i> splits into. $1 \leq \textit{nsplit} \leq n$.
<i>isplit</i>	INTEGER Array of size <i>n</i> The splitting points, at which <i>T</i> breaks up into blocks. The first block consists of rows/columns 1 to $\textit{isplit}(1)$, the second of rows/columns $\textit{isplit}(1)+1$ through $\textit{isplit}(2)$, etc., and the <i>nsplit</i> -th block consists of rows/columns $\textit{isplit}(\textit{nsplit}-1)+1$ through $\textit{isplit}(\textit{nsplit})=n$.
<i>m</i>	INTEGER The total number of eigenvalues (of all $L_i D_i L_i^T$) found.
<i>w</i>	REAL for <i>slarre2</i> DOUBLE PRECISION for <i>dlarre2</i> Array of size <i>n</i> The first <i>m</i> elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (<i>?larre2</i> may use the remaining <i>n-m</i> elements as workspace). Note that immediately after exiting this routine, only the eigenvalues from position <i>dol:dou</i> in <i>w</i> might rely on this processor when the eigenvalue computation is done in parallel.
<i>werr</i>	REAL for <i>slarre2</i> DOUBLE PRECISION for <i>dlarre2</i> Array of size <i>n</i> The error bound on the corresponding eigenvalue in <i>w</i> . Note that immediately after exiting this routine, only the uncertainties from position <i>dol:dou</i> in <i>werr</i> might rely on this processor when the eigenvalue computation is done in parallel.
<i>wgap</i>	REAL for <i>slarre2</i> DOUBLE PRECISION for <i>dlarre2</i> Array of size <i>n</i> The separation from the right neighbor eigenvalue in <i>w</i> .

The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this routine, only the gaps from position *dol:dou* in *wgap* might rely on this processor when the eigenvalue computation is done in parallel.

iblock

INTEGER Array of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock(i)=1* if eigenvalue *w(i)* belongs to the first block from the top, *iblock(i)=2* if *w(i)* belongs to the second block, and so on.

indexw

INTEGER Array of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw(i)= 10* and *iblock(i)=2* imply that the *i*-th eigenvalue *w(i)* is the 10th eigenvalue in block 2.

gers

REAL for *slarre2*

DOUBLE PRECISION for *dlarre2*

Array of size $2*n$

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers(2*i-1)*, *gers(2*i)*)).

pivmin

REAL for *slarre2*

DOUBLE PRECISION for *dlarre2*

The minimum pivot in the sturm sequence for *T*.

info

INTEGER

= 0: successful exit

> 0: A problem occurred in *?larre2*.

< 0: One of the called subroutines signaled an internal problem.

Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in *?larrd*.

=-2: Not enough internal iterations to find the base representation.

=-3: Problem in *?larrb* when computing the refined root representation for *?lasq2*.

=-4: Problem in *?larrb* when performing bisection on the desired part of the spectrum.

=-5: Problem in *?lasq2*

=-6: Problem in *?lasq2*

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larre2a

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```
call slarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
  isplit, m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam,
  pivmin, work, iwork, minrgp, info )
```

```
call dlarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
  isplit, m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam,
  pivmin, work, iwork, minrgp, info )
```

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , ?larre2a sets any "small" off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block's spectrum,
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

NOTE

The algorithm obtains a crude picture of all the wanted eigenvalues (as selected by *range*). However, to reduce work and improve scalability, only the eigenvalues *dol* to *dou* are refined. Furthermore, if the matrix splits into blocks, RRRs for blocks that do not contain eigenvalues from *dol* to *dou* are skipped. The DQDS algorithm (subroutine ?lasq2) is not used, unlike in the sequential case. Instead, eigenvalues are computed in parallel to some figures using bisection.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

range CHARACTER

- = 'A': ("All") all eigenvalues will be found.
- = 'V': ("Value") all eigenvalues in the half-open interval (*vl*, *vu*] will be found.
- = 'I': ("Index") the *il*-th through *iu*-th eigenvalues (of the entire matrix) will be found.

<i>n</i>	INTEGER The order of the matrix. $n > 0$.
<i>vl, vu</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> If <i>range</i> ='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <i>vl</i> , or greater than <i>vu</i> , will not be returned. $vl < vu$. If <i>range</i> ='I' or ='A', <code>?larre2a</code> computes bounds on the desired part of the spectrum.
<i>il, iu</i>	INTEGER If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$.
<i>d</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> Array of size <i>n</i> On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> .
<i>e</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> Array of size <i>n</i> The first (<i>n</i> -1) entries contain the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e</i> (<i>n</i>) need not be set.
<i>e2</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> Array of size <i>n</i> The first (<i>n</i> -1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix <i>T</i> ; <i>e2</i> (<i>n</i>) need not be set.
<i>rtol1, rtol2</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> Parameters for bisection. An interval [<i>left</i> , <i>right</i>] has converged if $right - left < \max(rtol1*gap, rtol2*\max(left , right))$
<i>spltol</i>	REAL for <code>slarre2a</code> DOUBLE PRECISION for <code>dlarre2a</code> The threshold for splitting.
<i>dol, dou</i>	INTEGER If the user wants to work on only a selected part of the representation tree, he can specify an index range <i>dol:dou</i> .

Otherwise, the setting $dol=1$, $dou=n$ should be applied.

Note that dol and dou refer to the order in which the eigenvalues are stored in w .

<i>work</i>	REAL for slarre2a DOUBLE PRECISION for dlarre2a Workspace array of size $6*n$
<i>iwork</i>	INTEGER Workspace array of size $5*n$
<i>minrgp</i>	REAL for slarre2a DOUBLE PRECISION for dlarre2a The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

OUTPUT Parameters

<i>vl, vu</i>	REAL for slarre2a DOUBLE PRECISION for dlarre2a If $range='V'$, the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to vl , or greater than vu , are not returned. $vl < vu$. If $range='I'$ or $range='A'$, ?larre2a computes bounds on the desired part of the spectrum.
<i>d</i>	The n diagonal elements of the diagonal matrices D_i .
<i>e</i>	e contains the subdiagonal elements of the unit bidiagonal matrices L_j . The entries $e(isplit(i))$, $1 \leq i \leq nsplit$, contain the base points σ_i on output.
<i>e2</i>	The entries $e2(isplit(i))$, $1 \leq i \leq nsplit$ have been set to zero.
<i>nsplit</i>	INTEGER The number of blocks T splits into. $1 \leq nsplit \leq n$.
<i>isplit</i>	INTEGER Array of size n The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, etc., and the $nsplit$ -th block consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.
<i>m</i>	INTEGER The total number of eigenvalues (of all $L_i D_i L_i^T$) found.
<i>needil, neediu</i>	INTEGER

The indices of the leftmost and rightmost eigenvalues of the root node RRR which are needed to accurately compute the relevant part of the representation tree.

*w*REAL for *slarre2a*DOUBLE PRECISION for *dlarre2a*Array of size *n*

The first *m* elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (*slarre2a* may use the remaining *n-m* elements as workspace).

Note that immediately after exiting this routine, only the eigenvalues from position *dol:dou* in *w* rely on this processor because the eigenvalue computation is done in parallel.

*werr*REAL for *slarre2a*DOUBLE PRECISION for *dlarre2a*Array of size *n*

The error bound on the corresponding eigenvalue in *w*.

Note that immediately after exiting this routine, only the uncertainties from position *dol:dou* in *werr* are reliable on this processor because the eigenvalue computation is done in parallel.

*wgap*REAL for *slarre2a*DOUBLE PRECISION for *dlarre2a*Array of size *n*

The separation from the right neighbor eigenvalue in *w*. The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this routine, only the gaps from position *dol:dou* in *wgap* are reliable on this processor because the eigenvalue computation is done in parallel.

*iblock*INTEGER Array of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock(i)*=1 if eigenvalue *w(i)* belongs to the first block from the top, *iblock(i)*=2 if *w(i)* belongs to the second block, and so on.

*indexw*INTEGER Array of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw(i)*= 10 and *iblock(i)*=2 imply that the *i*-th eigenvalue *w(i)* is the 10th eigenvalue in block 2.

*gers*REAL for *slarre2a*DOUBLE PRECISION for *dlarre2a*Array of size $2*n$

The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).

pivmin

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

The minimum pivot in the sturm sequence for T .

info

INTEGER

= 0: successful exit

> 0: A problem occurred in ?larre2a.

< 0: One of the called subroutines signaled an internal problem. Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in ?larrd2.

=-2: Not enough internal iterations to find base representation.

=-3: Problem in ?larrb2 when computing the refined root representation.

=-4: Problem in ?larrb2 when performing bisection on the desired part of the spectrum.

= -9 Problem: $m < dou-dol+1$, that is the code found fewer eigenvalues than it was supposed to.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrf2

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf2( n, d, l, ld, clstrt, clend, clmid1, clmid2, w, wgap, werr, trymid,
            spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf2( n, d, l, ld, clstrt, clend, clmid1, clmid2, w, wgap, werr, trymid,
            spdiam, clgapl, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

Description

Given the initial representation LDL^T and its cluster of close eigenvalues (in a relative measure), $w(clstrt), w(clstrt+1), \dots w(clend)$, ?larrf2 finds a new relatively robust representation $LDL^T - \sigma I = L_+D_+L_+^T$ such that at least one of the eigenvalues of $L_+D_+L_+^T$ is relatively isolated.

This is an enhanced version of ?larrf that also tries shifts in the middle of the cluster, should there be a large gap, in order to break large clusters into at least two pieces.

Input Parameters

n INTEGER

The order of the matrix (subblock, if the matrix was split).

d REAL for slarrf2

DOUBLE PRECISION for dlarrf2
 Array of size n
 The n diagonal elements of the diagonal matrix D .

l REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Array of size $n-1$
 The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L .

ld REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Array of size $n-1$
 The $(n-1)$ elements $l(i)*d(i)$.

$clstrt$ INTEGER
 The index of the first eigenvalue in the cluster.

$clend$ INTEGER
 The index of the last eigenvalue in the cluster.

$clmid1, clmid2$ INTEGER
 The index of a middle eigenvalue pair with large gap.

w REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Array of size $\geq (clend-clstrt+1)$
 The eigenvalue approximations of $LD L^T$ in ascending order. $w(clstrt)$ through $w(clend)$ form the cluster of relatively close eigenvalues.

$wgap$ REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Array of size $\geq (clend-clstrt+1)$
 The separation from the right neighbor eigenvalue in w .

$werr$ REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Array of size $\geq (clend-clstrt+1)$
 $werr$ contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in w .

$spdiam$ REAL for slarrf2
 DOUBLE PRECISION for dlarrf2
 Estimate of the spectral diameter obtained from the Gerschgorin intervals

<i>clgapl, clgapr</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.
<i>pivmin</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 The minimum pivot allowed in the Sturm sequence.
<i>work</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 Workspace array of size $2*n$

OUTPUT Parameters

<i>wgap</i>	Contains refined values of its input approximations. Very small gaps are unchanged.
<i>sigma</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 The shift (σ) used to form $L_+D_+L_+^T$.
<i>dplus</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 Array of size n The n diagonal elements of the diagonal matrix D_+ .
<i>lplus</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2 Array of size $n-1$ The first $(n-1)$ elements of <i>lplus</i> contain the subdiagonal elements of the unit bidiagonal matrix L_+ .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrv2

*Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L, D and the eigenvalues of $L*D*L^T$.*

Syntax

```
call slarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
```

```
call dlarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
```

Description

?larrv2 computes the eigenvectors of the tridiagonal matrix $T = LDL^T$ given L , D and approximations to the eigenvalues of LDL^T . The input eigenvalues should have been computed by [larre2a](#) or by previous calls to ?larrv2.

The major difference between the parallel and the sequential construction of the representation tree is that in the parallel case, not all eigenvalues of a given cluster might be computed locally. Other processors might "own" and refine part of an eigenvalue cluster. This is crucial for scalability. Thus there might be communication necessary before the current level of the representation tree can be parsed.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant when both eigenvalues and eigenvectors are computed (*stegr2b* parameter *jobz* = 'V'). ?larrv2 only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to $w(1)$ through $w(m)$, only the eigenvectors belonging to eigenvalues $w(dol)$ through $w(dou)$ are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* is true, the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to $dou - dol + 1$.
- ?larrv2 needs more workspace in *z* than the sequential *slarrv*. It is used to store the conformal embedding of the local representation tree.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>n</i>	INTEGER
	The order of the matrix. $n \geq 0$.
<i>vl, vu</i>	REAL for <i>slarrv2</i>
	DOUBLE PRECISION for <i>dlarrv2</i>

Lower and upper bounds of the interval that contains the desired eigenvalues. $v_l < v_u$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range. v_u is currently not used but kept as parameter in case needed.

<i>d</i>	<p>REAL for <code>slarrv2</code></p> <p>DOUBLE PRECISION for <code>dlarrv2</code></p> <p>Array of size n</p> <p>The n diagonal elements of the diagonal matrix d. On exit, d is overwritten.</p>
<i>l</i>	<p>REAL for <code>slarrv2</code></p> <p>DOUBLE PRECISION for <code>dlarrv2</code></p> <p>Array of size n</p> <p>The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L are in elements 1 to $n-1$ of l (if the matrix is not split.) At the end of each block is stored the corresponding shift as given by <code>?larre</code>. On exit, l is overwritten.</p>
<i>pivmin</i>	<p>DOUBLE PRECISION</p> <p>The minimum pivot allowed in the sturm sequence.</p>
<i>isplit</i>	<p>INTEGER</p> <p>Array of size n</p> <p>The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to <code>isplit(1)</code>, the second of rows/columns <code>isplit(1) + 1</code> through <code>isplit(2)</code>, etc.</p>
<i>m</i>	<p>INTEGER</p> <p>The total number of input eigenvalues. $0 \leq m \leq n$.</p>
<i>dol, dou</i>	<p>INTEGER</p> <p>If you want to compute only selected eigenvectors from all the eigenvalues supplied, you can specify an index range <code>dol:dou</code>. Or else the setting <code>dol=1, dou=m</code> should be applied. Note that <code>dol</code> and <code>dou</code> refer to the order in which the eigenvalues are stored in w. If you want to compute only selected eigenpairs, the columns <code>dol-1</code> to <code>dou+1</code> of the eigenvector space z contain the computed eigenvectors. All other columns of z are set to zero.</p> <p>If <code>dol > 1</code>, then <code>z(:,dol-1-zoffset)</code> is used.</p> <p>If <code>dou < m</code>, then <code>z(:,dou+1-zoffset)</code> is used.</p>
<i>needil, neediu</i>	<p>INTEGER</p> <p>Describe which are the left and right outermost eigenvalues that still need to be included in the computation. These indices indicate whether eigenvalues from other processors are needed to correctly compute the conformally embedded representation tree.</p>

When $dol \leq needil \leq neediu \leq dou$, all required eigenvalues are local to the processor and no communication is required to compute its part of the representation tree.

minrgp

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

rtol1, rtol2

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Parameters for bisection. An interval $[left, right]$ has converged if $right - left < \max(rtol1 * gap, rtol2 * \max(|left|, |right|))$

w

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size n

The first m elements of w contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array w from `?stegr2a` is expected here.) Furthermore, they are with respect to the shift of the corresponding root representation for their block.

werr

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size n

The first m elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in w .

wgap

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size n

The separation from the right neighbor eigenvalue in w .

*iblock*INTEGER Array of size n

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $iblock(i)=2$ if $w(i)$ belongs to the second block, and so on.

*indexw*INTEGER Array of size n

The indices of the eigenvalues within each block (submatrix). For example: $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10th eigenvalue in block 2.

gers

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size $2*n$

The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$). The Gerschgorin intervals should be computed from the original unshifted matrix.

Not used but kept as parameter for possible future use.

sdiam

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array of size n

The spectral diameters for all unreduced blocks.

ldz

INTEGER

The leading dimension of the array z . $ldz \geq 1$, and if `stegr2b` parameter `jobz = 'V'`, $ldz \geq \max(1, n)$.

work

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

(workspace) array of size $12*n$

iwork

(workspace) INTEGER Array of size $7*n$

vstart

LOGICAL

.TRUE. on initialization, set to .FALSE. afterwards.

finish

LOGICAL

A flag that indicates whether all eigenpairs have been computed.

maxcls

INTEGER

The largest cluster worked on by this processor in the representation tree.

ndepth

INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

INTEGER

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset

INTEGER

Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion.

OUTPUT Parameters

needil, neediu

w

Unshifted eigenvalues for which eigenvectors have already been computed.

werr

Contains refined values of its input approximations.

<i>wgap</i>	Contains refined values of its input approximations. Very small gaps are changed.
<i>z</i>	<p>REAL for <code>slarrv2</code></p> <p>DOUBLE PRECISION for <code>dlarrv2</code></p> <p>Array, dimension ($ldz, \max(1,m)$)</p> <p>If $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the input eigenvalues, with the i-th column of z holding the eigenvector associated with $w(i)$.</p> <p>In the distributed version, only a subset of columns is accessed, see <i>do1</i>, <i>dou</i>, and <i>zoffset</i>.</p>
<i>isuppz</i>	<p>INTEGER</p> <p>Array of size $2*\max(1,m)$</p> <p>The support of the eigenvectors in z, i.e., the indices indicating the non-zero elements in z. The i-th eigenvector is non-zero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$.</p>
<i>vstart</i>	.TRUE. on initialization, set to .FALSE. afterwards.
<i>finish</i>	A flag that indicates whether all eigenpairs have been computed.
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>info</i>	<p>INTEGER</p> <p>= 0: successful exit</p> <p>> 0: A problem occured in <code>?larrv2</code>.</p> <p>< 0: One of the called subroutines signaled an internal problem.</p> <p>Needs inspection of the corresponding parameter <i>info</i> for further information.</p> <p>=-1: Problem in <code>?larrb2</code> when refining a child's eigenvalues.</p> <p>=-2: Problem in <code>?larrf2</code> when computing the RRR of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter <i>minrgp</i> smaller and recompile. However, as the orthogonality of the computed vectors is proportional to $1/minrgp$, be aware that decreasing <i>minrgp</i> might be reduce precision.</p> <p>=-3: Problem in <code>?larrb2</code> when refining a single eigenvalue after the Rayleigh correction was rejected.</p> <p>= 5: The Rayleigh Quotient Iteration failed to converge to full accuracy.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
call slasorte(s, lds, j, out, info)
```

```
call dlasorte(s, lds, j, out, info)
```

Description

The ?lasorte routine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>s</i>	(local) REAL for slasorte DOUBLE PRECISION for dlasorte Array of size <i>lds</i> . On entry, a matrix already in Schur form.
<i>lds</i>	(local) INTEGER. On entry, the leading dimension of the array <i>s</i> ; unchanged on exit.
<i>j</i>	(local) INTEGER. On entry, the order of the matrix <i>S</i> ; unchanged on exit.
<i>out</i>	(local) REAL for slasorte DOUBLE PRECISION for dlasorte Array of size 2*j. The work buffer required by the routine.
<i>info</i>	(local) INTEGER. Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix <i>S</i> was not originally in Schur form. 0 indicates successful completion.

Output Parameters

<i>s</i>	On exit, the diagonal blocks of <i>S</i> have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.
<i>out</i>	Work buffer.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt2(id, n, d, key, info)
```

```
call dlasrt2(id, n, d, key, info)
```

Description

The ?lasrt2 routine is modified LAPACK routine ?lasrt, which sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . The size of *STACK* limits *n* to about 2^{32} .

Input Parameters

<i>id</i>	CHARACTER*1. = 'I': sort <i>d</i> in increasing order; = 'D': sort <i>d</i> in decreasing order.
<i>n</i>	INTEGER. The length of the array <i>d</i> .
<i>d</i>	REAL for slasrt2 DOUBLE PRECISION for dlasrt2. Array of size <i>n</i> . On entry, the array to be sorted.
<i>key</i>	INTEGER. Array of size <i>n</i> . On entry, <i>key</i> contains a key to each of the entries in <i>d</i> (). Typically, $key(i) = i$ for all <i>i</i> .

Output Parameters

<i>d</i>	On exit, <i>d</i> has been sorted into increasing order ($d(1) \leq \dots \leq d(n)$) or into decreasing order ($d(1) \geq \dots \geq d(n)$), depending on <i>id</i> .
----------	--

<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.
<i>key</i>	On exit, <i>key</i> is permuted in exactly the same manner as <i>d</i> was permuted from input to output. Therefore, if <i>key</i> (<i>i</i>) = <i>i</i> for all <i>i</i> on input, <i>d</i> (<i>i</i>) on output equals <i>d</i> (<i>key</i> (<i>i</i>)) on input.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
call sstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work,
lwork, iwork, liwork, dol, dou, zoffset, info )
```

```
call dstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work,
lwork, iwork, liwork, dol, dou, zoffset, info )
```

Description

?stegr2 computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*. It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version either when only eigenvalues are to be computed, or when only a single processor is used (the sequential-like case).

?stegr2 has been adapted from LAPACK's ?stegr. Please note the following crucial changes.

1. The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. ?stegr2 only computes the eigenpairs corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenpairs belonging to $w(1)$ through $w(m)$, only the eigenvectors belonging to eigenvalues $w(dol)$ through $w(dou)$ are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be fully accurate.
2. *m* is not the number of eigenvalues specified by *range*, but is $m = dou - dol + 1$. This concerns the case where only eigenvalues are computed, but on more than one processor. Thus, in this case *m* refers to the number of eigenvalues computed on this processor.
3. The arrays *w* and *z* might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>jobz</i>	<p>CHARACTER*1</p> <p>= 'N': Compute eigenvalues only;</p> <p>= 'V': Compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>CHARACTER*1</p> <p>= 'A': all eigenvalues will be found.</p> <p>= 'V': all eigenvalues in the half-open interval $(v_l, v_u]$ will be found.</p> <p>= 'I': the <i>il</i>-th through <i>iu</i>-th eigenvalues (of the entire matrix) will be found.</p>
<i>n</i>	<p>INTEGER</p> <p>The order of the matrix. $n \geq 0$.</p>
<i>d</i>	<p>REAL for <i>sstegr2</i></p> <p>DOUBLE PRECISION for <i>dstegr2</i></p> <p>Array of size <i>n</i></p> <p>On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>. Overwritten on exit.</p>
<i>e</i>	<p>REAL for <i>sstegr2</i></p> <p>DOUBLE PRECISION for <i>dstegr2</i></p> <p>Array of size <i>n</i></p> <p>On entry, the $(n-1)$ subdiagonal elements of the tridiagonal matrix <i>T</i> in elements 1 to <i>n</i>-1 of <i>e</i>. $e(n)$ need not be set on input, but is used internally as workspace. Overwritten on exit.</p>
<i>vl</i>	<p>REAL for <i>sstegr2</i></p> <p>DOUBLE PRECISION for <i>dstegr2</i></p>
<i>vu</i>	<p>REAL for <i>sstegr2</i></p> <p>DOUBLE PRECISION for <i>dstegr2</i></p> <p>If <i>range</i>='V', the lower and upper bounds of the interval to be searched for eigenvalues. $vl < vu$.</p> <p>Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il, iu</i>	<p>INTEGER</p> <p>If <i>range</i>='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned.</p> <p>$1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>Not referenced if <i>range</i> = 'A' or 'V'.</p>
<i>ldz</i>	<p>INTEGER</p>

The leading dimension of the array z . $ldz \geq 1$, and if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

nzc

INTEGER

The number of eigenvectors to be held in the array z , storing the matrix Z .

If $range = 'A'$, then $nzc \geq \max(1, n)$.

If $range = 'V'$, then $nzc \geq$ the number of eigenvalues in $(vl, vu]$.

If $range = 'I'$, then $nzc \geq iu - il + 1$.

If $nzc = -1$, then a workspace query is assumed; the routine calculates the number of columns of the matrix Z that are needed to hold the eigenvectors. This value is returned as the first entry of the z array, and no error message related to nzc is issued.

$lwork$

INTEGER

The size of the array $work$. $lwork \geq \max(1, 18 * n)$

if $jobz = 'V'$, and $lwork \geq \max(1, 12 * n)$ if $jobz = 'N'$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued.

$liwork$

INTEGER

The size of the array $iwork$. $liwork \geq \max(1, 10 * n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8 * n)$ if only the eigenvalues are to be computed.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued.

dol, dou

INTEGER

From the eigenvalues $w(1:m)$, only eigenvectors $Z(:, dol)$ to $Z(:, dou)$ are computed.

If $dol > 1$, then $Z(:, dol - 1 - zoffset)$ is used and overwritten.

If $dou < m$, then $Z(:, dou + 1 - zoffset)$ is used and overwritten.

$zoffset$

INTEGER

Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion

OUTPUT Parameters

m

INTEGER

Globally summed over all processors, m equals the total number of eigenvalues found. $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$. The local output equals $m = dou - dol + 1$.

w

REAL array of size n for `sstegr2`

DOUBLE PRECISION array of size n for `dstegr2`

Array of size n

The first m elements contain the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position $dol:dou$ are reliable on this processor because the eigenvalue computation is done in parallel. Other processors will hold reliable information on other parts of the w array. This information is communicated in the ScaLAPACK driver.

z

REAL for `sstegr2`

DOUBLE PRECISION for `dstegr2`

Array of size $(ldz, \max(1,m))$.

If $jobz = 'V'$, and if $info = 0$, then the first m columns of the matrix Z stored in z contain some of the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of Z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

Note: the user must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and can be computed with a workspace query by setting $nzc = -1$, see below.

$isuppz$

INTEGER array of size $2*\max(1,m)$

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. $isuppz$ is only set if $n > 2$.

$work$

On exit, if $info = 0$, $work(1)$ returns the optimal (and minimal) $lwork$.

$iwork$

On exit, if $info = 0$, $iwork(1)$ returns the optimal $liwork$.

$info$

INTEGER

On exit, $info$

= 0: successful exit

other: if $info = -i$, the i -th argument had an illegal value

if $info = 10X$, internal error in `?larre2`,

if $info = 20X$, internal error in `?larrv`.

Here, the digit $X = \text{ABS}(iinfo) < 10$, where $iinfo$ is the nonzero error code returned by `?larre2` or `?larrv`, respectively.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2a

Computes selected eigenvalues and initial representations needed for eigenvector computations.

Syntax

```
call sstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
call dstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
```

Description

?stegr2a computes selected eigenvalues and initial representations needed for eigenvector computations in ?stegr2b. It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version when both eigenvalues and eigenvectors are computed in parallel on multiple processors. For this case, ?stegr2a implements the first part of the MRRR algorithm, parallel eigenvalue computation and finding the root RRR. At the end of ?stegr2a, other processors might have a part of the spectrum that is needed to continue the computation locally. Once this eigenvalue information has been received by the processor, the computation can then proceed by calling the second part of the parallel MRRR algorithm, ?stegr2b.

Please note:

- The calling sequence has two additional integer parameters, (compared to LAPACK's `stegr`), these are `dol` and `dou` and should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case `jobz = 'V'`.

Globally invoked over all processors, ?stegr2a computes all the eigenvalues specified by `range`.

?stegr2a locally only computes the eigenvalues corresponding to eigenvalues `dol` through `dou` in `w`. (That is, instead of computing the eigenvectors belonging to $w(1)$ through $w(m)$, only the eigenvectors belonging to eigenvalues $w(dol)$ through $w(dou)$ are computed. In this case, only the eigenvalues `dol:dou` are guaranteed to be fully accurate.

- `m` is not the number of eigenvalues specified by `range`, but it is $m = dou - dol + 1$. Instead, `m` refers to the number of eigenvalues computed on this processor.
- While no eigenvectors are computed in ?stegr2a itself (this is done later in ?stegr2b), the interface

If `jobz = 'V'` then, depending on `range` and `dol, dou`, ?stegr2a might need more workspace in `z` than the original ?stegr. In particular, the arrays `w` and `z` might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

`jobz` CHARACTER*1
 = 'N': Compute eigenvalues only;
 = 'V': Compute eigenvalues and eigenvectors.

`range` CHARACTER*1

= 'A': all eigenvalues will be found.
 = 'V': all eigenvalues in the half-open interval $(v_l, v_u]$ will be found.
 = 'I': the i_l -th through i_u -th eigenvalues (of the entire matrix) will be found.

n INTEGER
 The order of the matrix. $n \geq 0$.

d REAL for sstegr2a
 DOUBLE PRECISION for dstegr2a
 Array of size n
 The n diagonal elements of the tridiagonal matrix T . Overwritten on exit.

e REAL for sstegr2a
 DOUBLE PRECISION for dstegr2a
 Array of size n
 On entry, the $(n-1)$ subdiagonal elements of the tridiagonal matrix T in elements 1 to $n-1$ of e . $e(n)$ need not be set on input, but is used internally as workspace. Overwritten on exit.

v_l, v_u REAL for sstegr2a
 DOUBLE PRECISION for dstegr2a
 If $range='V'$, the lower and upper bounds of the interval to be searched for eigenvalues. $v_l < v_u$.
 Not referenced if $range = 'A'$ or 'I'.

i_l, i_u INTEGER
 If $range='I'$, the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq i_l \leq i_u \leq n$, if $n > 0$.
 Not referenced if $range = 'A'$ or 'V'.

ldz INTEGER
 The leading dimension of the array z . $ldz \geq 1$, and if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

nzc INTEGER
 The number of eigenvectors to be held in the array z .
 If $range = 'A'$, then $nzc \geq \max(1, n)$.
 If $range = 'V'$, then $nzc \geq$ the number of eigenvalues in $(v_l, v_u]$.
 If $range = 'I'$, then $nzc \geq i_u - i_l + 1$.
 If $nzc = -1$, then a workspace query is assumed; the routine calculates the number of columns of the array z that are needed to hold the eigenvectors. This value is returned as the first entry of the z array, and no error message related to nzc is issued.

<i>lwork</i>	INTEGER The size of the array <i>work</i> . $lwork \geq \max(1, 18*n)$ if <i>jobz</i> = 'V', and $lwork \geq \max(1, 12*n)$ if <i>jobz</i> = 'N'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued.
<i>liwork</i>	INTEGER The size of the array <i>iwork</i> . $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed. If <i>liwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued.
<i>dol, dou</i>	INTEGER From all the eigenvalues $w(1:m)$, only eigenvalues $w(dol:dou)$ are computed.

OUTPUT Parameters

<i>m</i>	INTEGER Globally summed over all processors, <i>m</i> equals the total number of eigenvalues found. $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu-il+1$. The local output equals $m = dou - dol + 1$.
<i>w</i>	REAL for <i>sstegr2a</i> DOUBLE PRECISION for <i>dstegr2a</i> Array of size <i>n</i> The first <i>m</i> elements contain approximations to the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position <i>dol:dou</i> are reliable on this processor because the eigenvalue computation is done in parallel. The other entries outside <i>dol:dou</i> are very crude preliminary approximations. Other processors hold reliable information on these other parts of the <i>w</i> array. This information is communicated in the ScaLAPACK driver.
<i>z</i>	REAL for <i>sstegr2a</i> DOUBLE PRECISION for <i>dstegr2a</i> Array of size (<i>ldz</i> , $\max(1, m)$). <i>?stegr2a</i> does not compute eigenvectors, this is done in <i>?stegr2b</i> . The argument <i>z</i> as well as all related other arguments only appear to keep the interface consistent and to signal to the user that this subroutine is meant to be used when eigenvectors are computed.

<i>work</i>	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal (and minimal) <i>lwork</i> .
<i>iwork</i>	On exit, if <i>info</i> = 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>needil, neediu</i>	INTEGER The indices of the leftmost and rightmost eigenvalues needed to accurately compute the relevant part of the representation tree. This information can be used to find out which processors have the relevant eigenvalue information needed so that it can be communicated.
<i>inderr</i>	INTEGER <i>inderr</i> points to the place in the work space where the eigenvalue uncertainties (errors) are stored.
<i>nsplit</i>	INTEGER The number of blocks into which <i>T</i> splits. $1 \leq \textit{nsplit} \leq n$.
<i>pivmin</i>	REAL for <i>sstegr2a</i> DOUBLE PRECISION for <i>dstegr2a</i> The minimum pivot in the sturm sequence for <i>T</i> .
<i>scale</i>	REAL for <i>sstegr2a</i> DOUBLE PRECISION for <i>dstegr2a</i> The scaling factor for the tridiagonal <i>T</i> .
<i>wl, wu</i>	REAL for <i>sstegr2a</i> DOUBLE PRECISION for <i>dstegr2a</i> The interval (<i>wl</i> , <i>wu</i>] contains all the wanted eigenvalues. It is either given by the user or computed in ?larre2a .
<i>info</i>	INTEGER On exit, <i>info</i> = 0: successful exit other: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value if <i>info</i> = 10 <i>x</i> , internal error in ?larre2a , Here, the digit <i>x</i> = $\text{abs}(iinfo) < 10$, where <i>iinfo</i> is the nonzero error code returned by ?larre2a .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2b

From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.

Syntax

```
call sstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork,
dol, dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls,
ndepth, parity, zoffset, info )
```

```
call dstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork,
dol, dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls,
ndepth, parity, zoffset, info )
```

Description

?stegr2b should only be called after a call to ?stegr2a. From eigenvalues and initial representations computed by ?stegr2a, ?stegr2b computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors. It is potentially invoked multiple times on a given processor because the locally relevant representation tree might depend on spectral information that is "owned" by other processors and might need to be communicated.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case *jobz* = 'V'. ?stegr2b only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to $w(1)$ through $w(m)$, only the eigenvectors belonging to eigenvalues $w(dol)$ through $w(dou)$ are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* equals .TRUE., the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to $dou - dol + 1$.
- ?stegr2b needs more workspace in *z* than the sequential ?stegr. It is used to store the conformal embedding of the local representation tree.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

jobz CHARACTER*1
= 'N': Compute eigenvalues only;
= 'V': Compute eigenvalues and eigenvectors.

n INTEGER

	The order of the matrix. $n \geq 0$.
<i>d</i>	REAL for <i>sstegr2b</i> DOUBLE PRECISION for <i>dstegr2b</i> Array of size <i>n</i> The <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . Overwritten on exit.
<i>e</i>	REAL for <i>sstegr2b</i> DOUBLE PRECISION for <i>dstegr2b</i> Array of size <i>n</i> The (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix <i>T</i> in elements 1 to <i>n</i> -1 of <i>e</i> . <i>e</i> (<i>n</i>) need not be set on input, but is used internally as workspace. Overwritten on exit.
<i>m</i>	INTEGER The total number of eigenvalues found in <i>?stegr2a</i> . $0 \leq m \leq n$.
<i>w</i>	REAL for <i>sstegr2b</i> DOUBLE PRECISION for <i>dstegr2b</i> Array of size <i>n</i> The first <i>m</i> elements contain approximations to the selected eigenvalues in ascending order. Note that only the eigenvalues from the locally relevant part of the representation tree, that is all the clusters that include eigenvalues from <i>dol:dou</i> , are reliable on this processor. (It does not need to know about any others anyway.)
<i>ldz</i>	INTEGER The leading dimension of the array <i>z</i> . $ldz \geq 1$, and if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>nzc</i>	INTEGER The number of eigenvectors to be held in the array <i>z</i> , storing the matrix <i>Z</i> .
<i>lwork</i>	INTEGER The size of the array <i>work</i> . $lwork \geq \max(1, 18*n)$ if <i>jobz</i> = 'V', and $lwork \geq \max(1, 12*n)$ if <i>jobz</i> = 'N'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued.
<i>liwork</i>	INTEGER The size of the array <i>iwork</i> . $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued.

dol, dou

INTEGER

From the eigenvalues $w(1:m)$, only eigenvectors $Z(:,dol)$ to $Z(:,dou)$ are computed.

If $dol > 1$, then $Z(:,dol-1-zoffset)$ is used and overwritten.

If $dou < m$, then $Z(:,dou+1-zoffset)$ is used and overwritten.

needil, neediu

INTEGER

Describes which are the left and right outermost eigenvalues still to be computed. Initially computed by `?larre2a`, modified in the course of the algorithm.

pivmin

REAL for `sstegr2b`

DOUBLE PRECISION for `dstegr2b`

The minimum pivot in the sturm sequence for T .

scale

REAL for `sstegr2b`

DOUBLE PRECISION for `dstegr2b`

The scaling factor for T . Used for unscaling the eigenvalues at the very end of the algorithm.

wl, wu

REAL for `sstegr2b`

DOUBLE PRECISION for `dstegr2b`

The interval $(wl, wu]$ contains all the wanted eigenvalues.

vstart

LOGICAL

.TRUE. on initialization, set to .FALSE. afterwards.

finish

LOGICAL

Indicates whether all eigenpairs have been computed.

maxcls

INTEGER

The largest cluster worked on by this processor in the representation tree.

ndepth

INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

INTEGER

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset

INTEGER

Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion.

OUTPUT Parameters

<i>z</i>	<p>REAL for <code>sstegr2b</code></p> <p>DOUBLE PRECISION for <code>dstegr2b</code></p> <p>Array of size $(ldz, \max(1,m))$</p> <p>If $jobz = 'V'$, and if $info = 0$, then a subset of the first m columns of the matrix Z, stored in z, contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with $w(i)$.</p> <p>See <i>dol</i>, <i>dou</i> for more information.</p>
<i>isuppz</i>	<p>INTEGER array of size $2*\max(1,m)$.</p> <p>The support of the eigenvectors in z, i.e., the indices indicating the nonzero elements in z. The i-th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. <i>isuppz</i> is only set if $n > 2$.</p>
<i>work</i>	On exit, if $info = 0$, <i>work</i> (1) returns the optimal (and minimal) <i>lwork</i> .
<i>iwork</i>	On exit, if $info = 0$, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>needil, neediu</i>	Modified in the course of the algorithm.
<i>indwlc</i>	<p>REAL for <code>sstegr2b</code></p> <p>DOUBLE PRECISION for <code>dstegr2b</code></p> <p>Pointer into the workspace location where the local eigenvalue representations are stored. ("Local eigenvalues" are those relative to the individual shifts of the RRRs.)</p>
<i>vstart</i>	.TRUE. on initialization, set to .FALSE. afterwards.
<i>finish</i>	Indicates whether all eigenpairs have been computed
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>info</i>	<p>INTEGER</p> <p>On exit, <i>info</i></p> <p>= 0: successful exit</p> <p>other: if $info = -i$, the i-th argument had an illegal value</p> <p>if $info = 20x$, internal error in <code>?larrv2</code>.</p>

Here, the digit $x = \text{abs}(iinfo) < 10$, where `iinfo` is the nonzero error code returned by `?larrv2`

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

```
call sstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
```

```
call dstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
```

Description

The `?stein2` routine is a modified LAPACK routine `?stein`. It computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter `maxits` (currently set to 5).

Input Parameters

<code>n</code>	INTEGER. The order of the matrix T ($n \geq 0$).
<code>m</code>	INTEGER. The number of eigenvectors to be found ($0 \leq m \leq n$).
<code>d, e, w</code>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <code>d(*)</code> , of size n . The n diagonal elements of the tridiagonal matrix T . <code>e(*)</code> , of size n . The $(n-1)$ subdiagonal elements of the tridiagonal matrix T , in elements 1 to $n-1$. <code>e(n)</code> need not be set. <code>w(*)</code> , of size n . The first m elements of w contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array w from <code>?stebz</code> with <code>ORDER = 'B'</code> is expected here). The size of w must be at least $\max(1, n)$.
<code>iblock</code>	INTEGER. Array of size n . The submatrix indices associated with the corresponding eigenvalues in w ; <code>iblock(i) = 1</code> , if eigenvalue $w(i)$ belongs to the first submatrix from the top,

$iblock(i) = 2$, if eigenvalue $w(i)$ belongs to the second submatrix, etc. (The output array $iblock$ from `?stebz` is expected here).

isplit

INTEGER.

Array of size n .

The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second submatrix consists of rows/columns $isplit(1)+1$ through $isplit(2)$, etc. (The output array $isplit$ from `?stebz` is expected here).

orfac

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within $orfac * ||T||$ of each other are to be orthogonalized.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq \max(1, n)$.

work

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Workspace array of size $5n$.

iwork

INTEGER. Workspace array of size n .

Output Parameters

z

REAL for `sstein2`

DOUBLE PRECISION for `dstein2`

Array of size ldz by m .

The computed eigenvectors. The eigenvector associated with the eigenvalue $w(i)$ is stored in the i -th column of z . Any vector that fails to converge is set to its current iterate after $maxits$ iterations.

ifail

INTEGER.

Array of size m .

On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after $maxits$ iterations, then their indices are stored in the array *ifail*.

info

INTEGER.

$info = 0$, the exit is successful.

$info < 0$: if $info = -i$, the i -th had an illegal value.

$info > 0$: if $info = i$, then i eigenvectors failed to converge in $maxits$ iterations. Their indices are stored in the array *ifail*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dgbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
call sdbtf2(m, n, kl, ku, ab, ldab, info)
call ddbtf2(m, n, kl, ku, ab, ldab, info)
call cdbtf2(m, n, kl, ku, ab, ldab, info)
call zdbtf2(m, n, kl, ku, ab, ldab, info)
```

Description

The ?dgbtf2 routine computes an LU factorization of a general real/complex m -by- n band matrix A without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
ab	REAL for sdbtf2 DOUBLE PRECISION for ddbtf2 COMPLEX for cdbtf2 COMPLEX*16 for zdbtf2. Array of size $ldab$ by n . The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl of the array need not be set. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(kl+ku+1+i-j,j) = A(i,j)$ for $\max(1,j-ku) \leq i \leq \min(m,j+kl)$.
$ldab$	INTEGER. The leading dimension of the array ab . ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab	On exit, details of the factorization: U is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the <i>Application Notes</i> below for further details.
$info$	INTEGER. = 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value,

> 0: if $info = +i$, $U(i,i)$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:



The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of U , because of fill-in resulting from the row interchanges.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdbtrf(m, n, kl, ku, ab, ldab, info)
call ddbtrf(m, n, kl, ku, ab, ldab, info)
call cdbtrf(m, n, kl, ku, ab, ldab, info)
call zdbtrf(m, n, kl, ku, ab, ldab, info)
```

Description

This routine computes an LU factorization of a real m -by- n band matrix A without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
kl	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
ab	REAL for sdbtrf DOUBLE PRECISION for ddbtrf COMPLEX for cdbtrf

COMPLEX*16 for zdbtrf.

Array of size $ldab$ by n .

The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to k of the array need not be set. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(kl+ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.

$ldab$ INTEGER. The leading dimension of the array ab .
 $(ldab \geq 2kl + ku + 1)$

Output Parameters

ab On exit, details of the factorization: U is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the *Application Notes* below for further details.

$info$ INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument had an illegal value,
 > 0: if $info = +i$, $U(i,i)$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6, kl = 2, ku = 1$:



The routine does not use array elements marked *.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dttf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
call sdttrf(n, dl, d, du, info)
call ddttrf(n, dl, d, du, info)
call cdttrf(n, dl, d, du, info)
call zdttrf(n, dl, d, du, info)
```

Description

The `?dttrf` routine computes an LU factorization of a real or complex tridiagonal matrix A using elimination without partial pivoting.

The factorization has the form $A = L*U$, where L is a product of unit lower bidiagonal matrices and U is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n INTEGER. The order of the matrix A ($n \geq 0$).

dl, d, du REAL for `sdttrf`
 DOUBLE PRECISION for `ddttrf`
 COMPLEX for `cdttrf`
 COMPLEX*16 for `zdttrf`.

Arrays containing elements of A .

The array dl of size $(n-1)$ contains the sub-diagonal elements of A .

The array d of size n contains the diagonal elements of A .

The array du of size $(n-1)$ contains the super-diagonal elements of A .

Output Parameters

dl Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .

d Overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

du Overwritten by the $(n-1)$ elements of the first super-diagonal of U .

$info$ INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument had an illegal value,
 > 0: if $info = i$, $U(i,i)$ is exactly 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by `?dttrf`.

Syntax

```
call sdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

call `zdttrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)`

Description

The `?dttrsv` routine solves one of the following systems of linear equations:

$$L * X = B, L^T * X = B, \text{ or } L^H * X = B,$$

$$U * X = B, U^T * X = B, \text{ or } U^H * X = B$$

with factors of the tridiagonal matrix A from the LU factorization computed by `?dttrf`.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether to solve with L or U .
<i>trans</i>	CHARACTER. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $A * X = B$ is solved for X (no transpose). If <i>trans</i> = 'T', then $A^T * X = B$ is solved for X (transpose). If <i>trans</i> = 'C', then $A^H * X = B$ is solved for X (conjugate transpose).
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns in the matrix B ($nrhs \geq 0$).
<i>dl,d,du,b</i>	REAL for <code>sdttrsv</code> DOUBLE PRECISION for <code>ddttrsv</code> COMPLEX for <code>cdttrsv</code> COMPLEX*16 for <code>zdttrsv</code> . The array <i>dl</i> of size $(n - 1)$ contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A . The array <i>d</i> of size n contains n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array <i>du</i> of size $(n - 1)$ contains the $(n - 1)$ elements of the first super-diagonal of U . On entry, the array <i>b</i> of size $(ldb, nrhs)$ contains the right-hand side of matrix B .
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?pttrsv

Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the $L^*D^*L^H$ factorization computed by ?pttrf.

Syntax

```
call spttrsv(trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv(trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
```

Description

The ?pttrsv routine solves one of the triangular systems:

$L^T * X = B$, or $L * X = B$ for real flavors,

or

$L * X = B$, or $L^H * X = B$,

$U * X = B$, or $U^H * X = B$ for complex flavors,

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = L^*D^*L^H$ (computed by `spttrf/dpttrf`)

or

$A = U^H * D * U$ or $A = L^*D^*L^H$ (computed by `cpttrf/zpttrf`).

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.
Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:
If *uplo* = 'U', e is the superdiagonal of U , and $A = U^H * D * U$ or $A = L^*D^*L^H$;
if *uplo* = 'L', e is the subdiagonal of L , and $A = L^*D^*L^H$.
The two forms are equivalent, if A is real.

trans CHARACTER.
Specifies the form of the system of equations:
for real flavors:
if *trans* = 'N': $L * X = B$ (no transpose)
if *trans* = 'T': $L^T * X = B$ (transpose)
for complex flavors:
if *trans* = 'N': $U * X = B$ or $L * X = B$ (no transpose)
if *trans* = 'C': $U^H * X = B$ or $L^H * X = B$ (conjugate transpose).

<i>n</i>	INTEGER. The order of the tridiagonal matrix <i>A</i> . $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, that is, the number of columns of the matrix <i>B</i> . $nrhs \geq 0$.
<i>d</i>	REAL array of size <i>n</i> . The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization computed by ?pttrf .
<i>e</i>	COMPLEX array of size $(n-1)$. The $(n-1)$ off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by ?pttrf . See <i>uplo</i> .
<i>b</i>	COMPLEX array of size <i>ldb</i> by <i>nrhs</i> . On entry, the right hand side matrix <i>B</i> .
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	On exit, the solution matrix <i>X</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = $-i$, the <i>i</i> -th argument had an illegal value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
call ssteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

```
call dsteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

Description

The [?steqr2](#) routine is a modified version of LAPACK routine [?steqr](#). The [?steqr2](#) routine computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. [?steqr2](#) is modified from [?steqr](#) to allow each ScaLAPACK process running [?steqr2](#) to perform updates on a distributed matrix *Q*. Proper usage of [?steqr2](#) can be gleaned from examination of ScaLAPACK routine [p?syev](#).

Input Parameters

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I'.
--------------	----------------------------------

If `compz = 'N'`, the routine computes eigenvalues only. If `compz = 'I'`, the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T .

z must be initialized to the identity matrix by `p?laset` or `?laset` prior to entering this subroutine.

n INTEGER. The order of the matrix T ($n \geq 0$).

$d, e, work$ REAL for `ssteqr2`

DOUBLE PRECISION for `dsteqr2`

Arrays:

d contains the diagonal elements of T . The size of d must be at least $\max(1, n)$.

e contains the $(n-1)$ subdiagonal elements of T . The size of e must be at least $\max(1, n-1)$.

$work$ is a workspace array. The size of $work$ is $\max(1, 2*n-2)$. If `compz = 'N'`, then $work$ is not referenced.

z (local)

REAL for `ssteqr2`

DOUBLE PRECISION for `dsteqr2`

Array of global size n by n and of local size ldz by nr .

If `compz = 'V'`, then z contains the orthogonal matrix used in the reduction to tridiagonal form.

ldz INTEGER. The leading dimension of the array z . Constraints:

$ldz \geq 1$,

$ldz \geq \max(1, n)$, if eigenvectors are desired.

nr INTEGER. $nr = \max(1, \text{numroc}(n, nb, \text{myproc}, 0, nprocs))$.

If `compz = 'N'`, then nr is not referenced.

Output Parameters

d On exit, the eigenvalues in ascending order, if `info = 0`.

See also `info`.

e On exit, e has been destroyed.

z On exit, if `info = 0`, then,

if `compz = 'V'`, z contains the orthonormal eigenvectors of the original symmetric matrix, and if `compz = 'I'`, z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If `compz = 'N'`, then z is not referenced.

$info$ INTEGER.

$info = 0$, the exit is successful.

$info < 0$: if $info = -i$, the i -th had an illegal value.

$info > 0$: the algorithm has failed to find all the eigenvalues in a total of $30n$ iterations;

if $info = i$, then i elements of e have not converged to zero; on exit, d and e contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
p?labad	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
p?lachieee	s, d	Performs a simple check for the features of the IEEE standard. (C interface function).
p?lamch	s, d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s, d	Computes the position of the sign bit of a floating-point number. (C interface function).

See Also

[pxerbla](#) Error handling routine called by ScaLAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call pslabad(ictxt, small, large)
```

```
call pdlabad(ictxt, small, large)
```

Description

The [p?labad](#) routine takes as input the values computed by [p?lamch](#) for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by [p?lamch](#). This subroutine is needed because [p?lamch](#) does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

<i>ictxt</i>	(global) INTEGER. The BLACS context handle in which the computation takes place.
<i>small</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad. On entry, the underflow threshold as computed by p?lamch.
<i>large</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad. On entry, the overflow threshold as computed by p?lamch.

Output Parameters

<i>small</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lachieee

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

```
void pslachieee(int *isiieee, float *rmax, float *rmin);
void pdlachieee(int *isiieee, float *rmax, float *rmin);
```

Description

The p?lachieee routine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?lachieee may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

<i>rmax</i>	(local). REAL for pslachieee DOUBLE PRECISION for pdlachieee
-------------	--

rmin The overflow threshold (= ?lamch ('O')).
 (local).
 REAL for pslachkieee
 DOUBLE PRECISION for pdlachkieee
 The underflow threshold (= ?lamch ('U')).

Output Parameters

isieee (local). INTEGER.
 On exit, *isieee* = 1 implies that all the features of the IEEE standard that we rely on are implemented. On exit, *isieee* = 0 implies that some the features of the IEEE standard that we rely on are missing.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

val = pslamch(*ictxt*, *cmach*)

val = pdlamch(*ictxt*, *cmach*)

Description

The p?lamchroutine determines single precision machine parameters.

Input Parameters

ictxt (global). INTEGER. The BLACS context handle in which the computation takes place.

cmach (global) CHARACTER*1.
 Specifies the value to be returned by p?lamch:

- = 'E' or 'e', p?lamch := eps
- = 'S' or 's', p?lamch := sfmin
- = 'B' or 'b', p?lamch := base
- = 'P' or 'p', p?lamch := eps*base
- = 'N' or 'n', p?lamch := t
- = 'R' or 'r', p?lamch := rnd
- = 'M' or 'm', p?lamch := emin
- = 'U' or 'u', p?lamch := rmin
- = 'L' or 'l', p?lamch := emax
- = 'O' or 'o', p?lamch := rmax,

where

eps = relative machine precision

sfmin = safe minimum, such that $1/\text{sfmin}$ does not overflow

base = base of the machine

$\text{prec} = \text{eps} * \text{base}$

t = number of (base) digits in the mantissa

$\text{rnd} = 1.0$ when rounding occurs in addition, 0.0 otherwise

emin = minimum exponent before (gradual) underflow

$\text{rmin} = \text{underflow threshold} - \text{base}^{(\text{emin}-1)}$

$\text{emax} = \text{largest exponent before overflow}$

$\text{rmax} = \text{overflow threshold} - (\text{base}^{\text{emax}}) * (1-\text{eps})$

Output Parameters

val Value returned by the routine.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

```
void pslasnbt(int *ieflag);
```

```
void pdlasnbt(int *ieflag);
```

Description

The `p?lasnbt` routine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, `ieflag = 0` is returned.

Output Parameters

ieflag INTEGER.

This flag indicates the position of the signbit of any single/double precision floating point number.

ieflag = 0, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.

ieflag = 1 indicates that the signbit is the 32-nd bit for a single precision routine.

In the case of a double precision routine:

ieflag = 1 indicates that the signbit is the 32-nd bit (Big Endian).

`ieflag = 2` indicates that the signbit is the 64-th bit (Little Endian).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Matrix Redistribution/Copy Routines

This section describes ScaLAPACK redistribution/copy routines. Summary information about these routines is given in the following table:

ScaLAPACK Redistribution/Copy Routines

Routine Name	Data Types	Description
<code>p?gemr2d</code>	<code>s, d, c, z, i</code>	Copies a submatrix from one general rectangular matrix to another.
<code>p?trmr2d</code>	<code>s, d, c, z, i</code>	Copies a submatrix from one trapezoidal matrix to another.

See Also

`pxerbla` Error handling routine called by ScaLAPACK routines.

`p?gemr2d`

Copies a submatrix from one general rectangular matrix to another.

Syntax

```
call psgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pcgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pzgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pigemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

Description

The `p?gemr2d` routine copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B . It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With `p?trmr2d`, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context A is disjoint from context B . The general rules for which parameters need to be set are:

- All calling processes must have the correct m and n .
- Processes in context A must correctly define all parameters describing A .
- Processes in context B must correctly define all parameters describing B .
- Processes which are not members of context A must pass `ctxt_a = -1` and need not set other parameters describing A .

- Processes which are not members of context B must pass $ctxt_b = -1$ and need not set other parameters describing B .

Because of its generality, `p?gemr2d` can be used for many operations not usually associated with copy routines. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with $dtype_ = 1$.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

m	(global) INTEGER. The number of rows of matrix A to be copied ($m \geq 0$).
n	(global) INTEGER. The number of columns of matrix A to be copied ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $sub(B)$ ($nrhs \geq 0$).
a	(local) REAL for <code>psgemr2d</code> DOUBLE for <code>pdgemr2d</code> COMPLEX for <code>pcgemr2d</code> DOUBLE COMPLEX for <code>pzgemr2d</code> INTEGER for <code>pigemr2d</code> . Pointer into the local memory to array of size $lld_aby \text{ LOCC}(ja+n-1)$ containing the source matrix A .
ia, ja	(global) INTEGER. The row and column indices in the array A indicating the first row and the first column, respectively, of the submatrix of A to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.
$desca$	(global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A . Only $dtype_a = 1$ is supported, so $dlen_ = 9$. If the calling process is not part of the context of A , $ctxt_a$ must be equal to -1 .

<i>ib, jb</i>	(global) INTEGER. The row and column indices in the array <i>B</i> indicating the first row and the first column, respectively, of the submatrix <i>B</i> to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . Only <i>dtype_b</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>B</i> , <i>ctxt_b</i> must be equal to -1.
<i>ictxt</i>	(global) INTEGER. The context encompassing at least the union of all processes in context <i>A</i> and context <i>B</i> . All processes in the context <i>ictxt</i> must call this routine, even if they do not own a piece of either matrix.

Output Parameters

<i>b</i>	REAL for psgemr2d DOUBLE for pdgemr2d COMPLEX for pcgemr2d DOUBLE COMPLEX for pzgemr2d INTEGER for pigemr2d. Pointer into the local memory to array of size <i>lld_b</i> by <i>LOCc(jb+nrhs-1)</i> . Overwritten by the submatrix from <i>A</i> .
----------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trmr2d

Copies a submatrix from one trapezoidal matrix to another.

Syntax

```
call pstrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdtrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pctrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pztrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pitrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

Description

The *p?trmr2d* routine copies the indicated matrix or submatrix of *A* to the indicated matrix or submatrix of *B*. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With *p?gemr2d*, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix *A* in context *A* (distributed over process grid *A*) and copy it to a matrix or submatrix *B* in context *B* (distributed over process grid *B*).

The `p?trmr2d` routine assumes the matrix or submatrix to be trapezoidal. Only the upper or lower part is copied, and the other part is unchanged.

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context *A* is disjoint from context *B*. The general rules for which parameters need to be set are:

- All calling processes must have the correct *m* and *n*.
- Processes in context *A* must correctly define all parameters describing *A*.
- Processes in context *B* must correctly define all parameters describing *B*.
- Processes which are not members of context *A* must pass `ctxt_a = -1` and need not set other parameters describing *A*.
- Processes which are not members of context *B* must pass `ctxt_b = -1` and need not set other parameters describing *B*.

Because of its generality, `p?trmr2d` can be used for many operations not usually associated with copy routines. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with `dtype_ = 1`.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

uplo (global) CHARACTER*1. Specifies whether to copy the upper or lower part of the matrix or submatrix.

`uplo = 'U'` Copy the upper triangular part.

`uplo = 'L'` Copy the lower triangular part.

diag (global) CHARACTER*1. Specifies whether to copy the diagonal of the matrix or submatrix.

`diag = 'U'` Do not copy the diagonal.

`diag = 'N'` Copy the diagonal.

<i>m</i>	(global) INTEGER. The number of rows of matrix <i>A</i> to be copied ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns of matrix <i>A</i> to be copied ($n \geq 0$).
<i>a</i>	(local) REAL for pstrmr2d DOUBLE for pdtrmr2d COMPLEX for pctrmr2d DOUBLE COMPLEX for pztrmr2d INTEGER for pitrmr2d. Pointer into the local memory to array of size <i>lld_a</i> by <i>LOCc(ja+n-1)</i> containing the source matrix <i>A</i> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the array <i>A</i> indicating the first row and the first column, respectively, of the submatrix of <i>A</i> to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . Only <i>dtype_a</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>A</i> , <i>ctxt_a</i> must be equal to -1.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the array <i>B</i> indicating the first row and the first column, respectively, of the submatrix <i>B</i> to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.
<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . Only <i>dtype_b</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>B</i> , <i>ctxt_b</i> must be equal to -1.
<i>ictxt</i>	(global) INTEGER. The context encompassing at least the union of all processes in context <i>A</i> and context <i>B</i> . All processes in the context <i>ictxt</i> must call this routine, even if they do not own a piece of either matrix.

Output Parameters

<i>b</i>	REAL for pstrmr2d DOUBLE for pdtrmr2d COMPLEX for pctrmr2d DOUBLE COMPLEX for pztrmr2d INTEGER for pitrmr2d. Pointer into the local memory to array of size <i>lld_b</i> by <i>LOCc(jb+nrhs-1)</i> .
----------	---

Overwritten by the submatrix from A .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Sparse Solver Routines

Intel® Math Kernel Library (Intel® MKL) provides user-callable sparse solver software to solve real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian square sparse linear system of algebraic equations.

The terms and concepts required to understand the use of the Intel MKL sparse solver routines are discussed in the [Appendix A "Linear Solvers Basics"](#). If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

This chapter describes

- the direct sparse solver based on PARDISO* which is referred to here as [Intel MKL PARDISO](#);
- the alternative interface for the direct sparse solver which is referred to here as the [DSS interface](#);
- [iterative sparse solvers \(ISS\)](#) based on the reverse communication interface (RCI);
- and [two preconditioners](#) based on the incomplete LU factorization technique.

Intel MKL PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel MKL PARDISO solver.

The Intel MKL PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [[Schenk00-2](#)]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [[Schenk00](#), [Schenk01](#), [Schenk02](#), [Schenk03](#)]. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

The following table lists the names of the Intel MKL PARDISO routines and describes their general use.

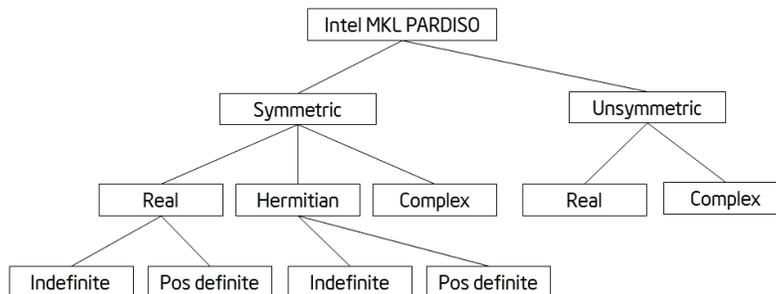
Intel MKL PARDISO Routines

Routine	Description
<code>pardisoinit</code>	Initializes Intel MKL PARDISO with default parameters depending on the matrix type.
<code>pardiso</code>	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.
<code>pardiso_64</code>	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.
<code>pardiso_getenv</code>	Retrieves additional values from the Intel MKL PARDISO handle.
<code>pardiso_setenv</code>	Sets additional values in the Intel MKL PARDISO handle.
<code>mkl_pardiso_pivot</code>	Replaces routine which handles Intel MKL PARDISO pivots with user-defined routine.
<code>pardiso_getdiag</code>	Returns diagonal elements of initial and factorized matrix.

Routine	Description
<code>pardiso_handle_store</code>	Store internal structures from <code>pardiso</code> to a file.
<code>pardiso_handle_restore</code>	Restore <code>pardiso</code> internal structures from a file.
<code>pardiso_handle_delete</code>	Delete files with <code>pardiso</code> internal structure data.
<code>pardiso_handle_store_64</code>	Store internal structures from <code>pardiso_64</code> to a file.
<code>pardiso_handle_restore_64</code>	Restore <code>pardiso_64</code> internal structures from a file.
<code>pardiso_handle_delete_64</code>	Delete files with <code>pardiso_64</code> internal structure data.

The Intel MKL PARDISO solver supports a wide range of real and complex sparse matrix types (see [the figure below](#)).

Sparse Matrices That Can Be Solved with the Intel MKL PARDISO Solver



The Intel MKL PARDISO solver performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
- termination to release all internal solver memory.

You can find code examples that use Intel MKL PARDISO routines to solve systems of linear equations in the `examples` folder of the Intel MKL installation directory:

- `examples/solverf/source`

Supported Matrix Types

The analysis steps performed by Intel MKL PARDISO depend on the structure of the input matrix A .

Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (both included with Intel MKL), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch-Kaufman pivoting for symmetric indefinite matrices. An approximation of X is found by forward and backward substitution and optional iterative refinement.

Whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block, the coefficient matrix is perturbed. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore, for a large set of matrices from different applications areas, this method is as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another method of improving the pivoting accuracy is to use symmetric weighted matching algorithms. These algorithms identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. These algorithms are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and optional iterative refinement.

Nonsymmetric Matrices

The solver first computes a nonsymmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$$QLUR = PP_{MPS}D_rAD_cP$$

with supernode pivoting matrices Q and R . When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of

$$\text{alpha} = \text{eps} * ||A2||_{\text{inf}},$$

where eps is the machine precision, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $||A2||_{\text{inf}}$ is the infinity norm of A . Any tiny pivots encountered during elimination are set to the sign $(1_{IT}) * \text{eps} * ||A2||_{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Sparse Data Storage

Sparse data storage in Intel MKL PARDISO follows the 3-array variation of the compressed sparse row (CSR3) format described in [Three Array Variation of CSR Format](#) or the three-array variation of the block compressed sparse row (BSR3) format described in [Three Array Variation of BSR Format](#). The Intel MKL PARDISO parameter ja is used for the *columns* array, ia is used for *rowIndex*, and a is used for *values*. The algorithms in Intel MKL PARDISO require column indices ja to be in increasing order per row and that the diagonal element in each row be present for any structurally symmetric matrix. For symmetric or nonsymmetric matrices the diagonal elements which are equal to zero are not necessary. For BSR3 format, use $iparm(37)$ to specify the block size.

CAUTION

Intel MKL PARDISO column indices ja must be in increasing order per row. You can validate the sparse matrix structure with the matrix checker ([*iparm*\(27\)](#))

NOTE

While the presence of zero diagonal elements for symmetric matrices is not required, you should explicitly set zero diagonal elements for symmetric matrices. Otherwise, Intel MKL PARDISO creates internal copies of arrays ia , ja , and a full of diagonal elements, which require additional memory and computational time. However, the memory and time required the diagonal elements in internal arrays is usually not significant compared to the memory and the time required to factor and solve the matrix.

Storage of Matrices

By default, Intel MKL PARDISO stores matrices in RAM. However, you can specify that Intel MKL PARDISO store matrices on disk by setting [*iparm*\(60\)](#). This is referred to as in-core (IC) and out-of-core (OOC), respectively.

OOC parameters can be set in a configuration file. You can set the path to this file and its name using environmental variables `MKL_PARDISO_OOC_CFG_PATH` and `MKL_PARDISO_OOC_CFG_FILE_NAME`.

These variables specify the path and filename as follows:

`<MKL_PARDISO_OOC_CFG_PATH>/<MKL_PARDISO_OOC_CFG_FILE_NAME>` for Linux* OS and OS X*, and
`<MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>` for Windows* OS.

By default, the name of the file is `pardiso_ooc.cfg` and it is placed in the current directory.

All temporary data files can be deleted or stored when the calculations are completed in accordance with the value of the environmental variable `MKL_PARDISO_OOC_KEEP_FILE`. If it is not set or if it is set to 1, then all files are deleted. If it is set to 0, then all files are stored.

By default, the OOC version of Intel MKL PARDISO uses the current directory for storing data, and all work arrays associated with the matrix factors are stored in files named `ooc_temp` with different extensions. These default values can be changed by using the environmental variable `MKL_PARDISO_OOC_PATH`.

To set the environmental variables `MKL_PARDISO_OOC_MAX_CORE_SIZE`, `MKL_PARDISO_OOC_MAX_SWAP_SIZE`, `MKL_PARDISO_OOC_KEEP_FILE`, and `MKL_PARDISO_OOC_PATH`, create the configuration file with the following lines:

```
MKL_PARDISO_OOC_PATH = <path>\ooc_file
MKL_PARDISO_OOC_MAX_CORE_SIZE = N
MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

where `<path>` is the directory for storing data, `ooc_file` is the file name without any extension, N is the maximum size of RAM in megabytes available for Intel MKL PARDISO (default value is 2000 MB), K is the maximum swap size in megabytes available for Intel MKL PARDISO (default value is 0 MB). Do not set $N+K$ greater than the size of the RAM plus the size of the swap. Be sure to allow enough free memory for the operating system and any other processes which are necessary.

CAUTION

The maximum length of the path lines in the configuration files is 1000 characters.

Alternatively the environment variables can be set via command line.

For Linux* OS and OS X*:

```
export MKL_PARDISO_OOC_PATH = <path>/ooc_file
export MKL_PARDISO_OOC_MAX_CORE_SIZE = N
export MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

For Windows* OS:

```
set MKL_PARDISO_OOC_PATH = <path>\ooc_file
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

NOTE

The values specified in a command line have higher priorities: if a variable is changed in the configuration file and in the command line, OOC version of Intel MKL PARDISO uses only the value defined in the command line.

Direct-Iterative Preconditioning for Nonsymmetric Linear Systems

The solver uses a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but with an identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. Intel MKL PARDISO uses a numerical factorization and applies the factors in a preconditioned Krylow-Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to nonsymmetric matrices in Intel MKL PARDISO. You can select the method using the *iparm*(4) input parameter. The *iparm*(20) parameter returns the error status after running Intel MKL PARDISO.

Single and Double Precision Computations

Intel MKL PARDISO solves tasks using single or double precision. Each precision has its benefits and drawbacks. Double precision variables have more digits to store value, so the solver uses more memory for keeping data. But this mode solves matrices with better accuracy, which is especially important for input matrices with large condition numbers.

Single precision variables have fewer digits to store values, so the solver uses less memory than in the double precision mode. Additionally this mode usually takes less time. But as computations are made less precisely, only some systems of equations can be solved accurately enough using single precision.

Separate Forward and Backward Substitution

The solver execution step (see *parameter**phase* = 33 below) can be divided into two or three separate substitutions: forward, backward, and possible diagonal. This separation can be explained by the examples of solving systems with different matrix types.

A real symmetric positive definite matrix A (*mtype* = 2) is factored by Intel MKL PARDISO as $A = L^*L^T$. In this case the solution of the system $A*x=b$ can be found as sequence of substitutions: $L*y=b$ (forward substitution, *phase* =331) and $L^T*x=y$ (backward substitution, *phase* =333).

A real nonsymmetric matrix A (*mtype* = 11) is factored by Intel MKL PARDISO as $A = L*U$. In this case the solution of the system $A*x=b$ can be found by the following sequence: $L*y=b$ (forward substitution, *phase* =331) and $U*x=y$ (backward substitution, *phase* =333).

Solving a system with a real symmetric indefinite matrix A (*mtype* = -2) is slightly different from the cases above. Intel MKL PARDISO factors this matrix as $A=LDL^T$, and the solution of the system $A*x=b$ can be calculated as the following sequence of substitutions: $L*y=b$ (forward substitution, *phase* =331), $D*v=y$

(diagonal substitution, *phase* =332), and finally $L^T * x = v$ (backward substitution, *phase* =333). Diagonal substitution makes sense only for symmetric indefinite matrices (*mtype* = -2, -4, 6). For matrices of other types a solution can be found as described in the first two examples.

CAUTION

The number of refinement steps (*iparm*(8)) must be set to zero if a solution is calculated with separate substitutions (*phase* = 331, 332, 333), otherwise Intel MKL PARDISO produces the wrong result.

NOTE

Different pivoting (*iparm*(21)) produces different LDL^T factorization. Therefore results of forward, diagonal and backward substitutions with diagonal pivoting can differ from results of the same steps with Bunch-Kaufman pivoting. Of course, the final results of sequential execution of forward, diagonal and backward substitution are equal to the results of the full solving step (*phase*=33) regardless of the pivoting used.

Callback Function for Pivoting Control

In-core Intel MKL PARDISO allows you to control pivoting with a callback routine, `mkl_pardiso_pivot`. You can then use the `pardiso_getdiag` routine to access the diagonal elements. Set *iparm*(56) to 1 in order to use the callback functionality.

pardiso

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
call pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl,
b, x, error)
```

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

The routine `pardiso` calculates the solution of a set of sparse linear equations

$$A * X = B$$

with single or multiple right-hand sides, using a parallel LU , LDL , or LL^T factorization, where A is an n -by- n matrix, and X and B are n -by-*nrhs* vectors or matrices.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice

effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters**NOTE**

The types given for parameters in this section are specified in FORTRAN 77 notation. See [Intel MKL PARDISO Parameters in Tabular Form](#) for detailed description of types of Intel MKL PARDISO parameters in Fortran 90 notation.

pt

INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with size of 64.

Handle to internal data structure. The entries must be set to zero prior to the first call to `pardiso`. Unique for factorization.

CAUTION

After the first call to `pardiso` do not directly modify *pt*, as that could cause a serious memory leak.

Use the [`pardiso_handle_store`](#) or [`pardiso_handle_store_64`](#) routine to store the content of *pt* to a file. Restore the contents of *pt* from the file using [`pardiso_handle_restore`](#) or [`pardiso_handle_restore_64`](#). Use `pardiso_handle_store` and `pardiso_handle_restore` with `pardiso`, and `pardiso_handle_store_64` and `pardiso_handle_restore_64` with `pardiso_64`.

maxfct

INTEGER

Maximum number of factors with identical sparsity structure that must be kept in memory at the same time. In most applications this value is equal to 1. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data structure management of the solver.

`pardiso` can process several matrices with an identical matrix sparsity pattern and it can store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum

INTEGER

Indicates the actual matrix for the solution phase. With this scalar you can define which matrix to factorize. The value must be: $1 \leq mnum \leq maxfct$.

In most applications this value is 1.

mtype

INTEGER

Defines the matrix type, which influences the pivoting method. The Intel MKL PARDISO solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase

INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Intel MKL PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement

This phase can be divided into two or three separate substitutions: forward, backward, and diagonal (see [Separate Forward and Backward Substitution](#)).

- Memory release phase (*phase*= 0 or *phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement

<i>phase</i>	Solver Execution Steps
33	Solve, iterative refinement
331	like <i>phase</i> =33, but only forward substitution
332	like <i>phase</i> =33, but only diagonal substitution (if available)
333	like <i>phase</i> =33, but only backward substitution
0	Release internal memory for <i>L</i> and <i>U</i> matrix number <i>mnum</i>
-1	Release all internal memory for all matrices

If *iparm*(36) = 0, phases 331, 332, and 333 perform this decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & U_{22} \end{bmatrix}$$

If *iparm*(36) = 2, phases 331, 332, and 333 perform a different decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & I \end{bmatrix}$$

You can supply a custom implementation for phase 332 instead of calling *pardiso*. For example, it can be implemented with dense LAPACK functionality. Custom implementation also allows you to substitute the matrix *S* with your own.

NOTE

For very large Schur complement matrices use LAPACK functionality to compute the Schur complement vector instead of the Intel MKL PARDISO phase 332 implementation.

n

INTEGER

Number of equations in the sparse linear systems of equations $A^*X = B$.
Constraint: $n > 0$.

a

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

Array. Contains the non-zero elements of the coefficient matrix A corresponding to the indices in ja . The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of ja for each row.

For CSR3 format, the size of a is the same as that of ja . Refer to the *values* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format the size of a is the size of ja multiplied by the square of the block size. Refer to the *values* array description in [Three Array Variation of BSR Format](#) for more details.

ia

INTEGER

Array, size $(n+1)$.

For CSR3 format, $ia(i)$ ($i \leq n$) points to the first column index of row i in the array ja . That is, $ia(i)$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zero elements in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format, $ia(i)$ ($i \leq n$) points to the first column index of row i in the array ja . That is, $ia(i)$ gives the index of the element in array a that contains the first non-zero block from row i of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zero blocks in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of BSR Format](#) for more details.

The array ia is accessed in all phases of the solution process.

Indexing of ia is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm(35)*.

ja

INTEGER

For CSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal elements are stored (even if they are zeros) in the list of non-zero elements in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of CSR Format](#).

For BSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal blocks are stored (even if they are zeros) in the list of non-zero blocks in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of BSR Format](#).

The array ja is accessed in all phases of the solution process.

Indexing of *ja* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm(35)*.

perm

INTEGER

Array, size (*n*). Depending on the value of *iparm(5)* and *iparm(31)*, either holds the permutation vector of size *n* or specifies elements used for computing a partial solution.

- If *iparm(5) = 1*, *iparm(31) = 0*, and *iparm(36) = 0*, *perm* specifies the fill-in reducing ordering to the solver. Let *A* be the original matrix and $C = P^*A^*P^T$ be the permuted matrix. Row (column) *i* of *C* is the *perm(i)* row (column) of *A*. The array *perm* is also used to return the permutation vector calculated during fill-in reducing ordering stage.

NOTE

Be aware that setting *iparm(5) = 1* prevents use of a parallel algorithm for the solve step.

- If *iparm(5) = 2*, *iparm(31) = 0*, and *iparm(36) = 0*, the permutation vector computed in phase 11 is returned in the *perm* array.
- If *iparm(5) = 0*, *iparm(31) > 0*, and *iparm(36) = 0*, *perm* specifies elements of the right-hand side to use or of the solution to compute for a partial solution.
- If *iparm(5) = 0*, *iparm(31) = 0*, and *iparm(36) > 0*, *perm* specifies elements for a Schur complement.

See *iparm(5)* and *iparm(31)* for more details.

Indexing of *perm* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm(35)*.

nrhs

INTEGER

Number of right-hand sides that need to be solved for.

iparm

INTEGER

Array, size (64). This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver.

See [pardiso iparm Parameter](#) for more details about the *iparm* parameters.

msglvl

INTEGER

Message level information. If *msglvl = 0* then *pardiso* generates no output, if *msglvl = 1* the solver prints statistical information to the screen.

b

DOUBLE PRECISION - for real types of matrices (*mtype=1, 2, -2 and 11*) and for double precision Intel MKL PARDISO (*iparm(28)=0*)

REAL - for real types of matrices (*mtype=1, 2, -2 and 11*) and for single precision Intel MKL PARDISO (*iparm(28)=1*)

DOUBLE COMPLEX - for complex types of matrices (*mtype=3, 6, 13, 14 and -4*) and for double precision Intel MKL PARDISO (*iparm(28)=0*)

COMPLEX - for complex types of matrices ($mtype=3, 6, 13, 14$ and -4) and for single precision Intel MKL PARDISO ($iparm(28)=1$)

Array, size $(n, nrhs)$. On entry, contains the right-hand side vector/matrix B , which is placed in memory contiguously. The $b(i+(k-1) \times nrhs)$ element must hold the i -th component of k -th right-hand side vector. Note that b is only accessed in the solution phase.

Output Parameters

(See also [Intel MKL PARDISO Parameters in Tabular Form.](#))

<i>pt</i>	Handle to internal data structure.
<i>perm</i>	See the Input Parameter description of the <i>perm</i> array.
<i>iparm</i>	On output, some <i>iparm</i> values report information such as the numbers of non-zero elements in the factors. See pardiso iparm Parameter for more details about the <i>iparm</i> parameters.
<i>b</i>	On output, the array is replaced with the solution if $iparm(6) = 1$.
<i>x</i>	DOUBLE PRECISION - for real types of matrices ($mtype=1, 2, -2$ and 11) and for double precision Intel MKL PARDISO ($iparm(28)=0$) REAL - for real types of matrices ($mtype=1, 2, -2$ and 11) and for single precision Intel MKL PARDISO ($iparm(28)=1$) DOUBLE COMPLEX - for complex types of matrices ($mtype=3, 6, 13, 14$ and -4) and for double precision Intel MKL PARDISO ($iparm(28)=0$) COMPLEX - for complex types of matrices ($mtype=3, 6, 13, 14$ and -4) and for single precision Intel MKL PARDISO ($iparm(28)=1$) Array, size $(n, nrhs)$. If $iparm(6)=0$ it contains solution vector/matrix X , which is placed contiguously in memory. The $x(i+(k-1) \times n)$ element must hold the i -th component of the k -th solution vector. Note that x is only accessed in the solution phase.

error INTEGER
The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	zero pivot, numerical factorization or iterative refinement problem
-5	unclassified (internal) error

error	Information
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files
-12	(<code>pardiso_64</code> only) <code>pardiso_64</code> called from 32-bit library

pardisoinit

Initialize Intel MKL PARDISO with default parameters in accordance with the matrix type.

Syntax

```
call pardisoinit (pt, mtype, iparm)
```

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function initializes Intel MKL PARDISO internal address pointer *pt* with zero values (as needed for the very first call of `pardiso`) and sets default *iparm* values in accordance with the matrix type. Intel MKL supplies the `pardisoinit` routine to be compatible with PARDISO 3.2 or lower.

NOTE

An alternative way to set default *iparm* values is to call `pardiso` in the analysis phase with `iparm(1)=0`. In this case you must initialize the internal address pointer *pt* with zero values manually.

NOTE

The `pardisoinit` routine initializes only the in-core version of Intel MKL PARDISO. Switching on the out-of core version of Intel MKL PARDISO as well as changing default *iparm* values can be done after the call to `pardisoinit` but before the first call to `pardiso`.

Input Parameters

<i>perm</i>	INTEGER
	Ignored.

NOTE

Parameters types in this section are specified in FORTRAN 77 notation. See [Intel MKL PARDISO Parameters in Tabular Form](#) section for detailed description of types of Intel MKL PARDISO parameters in Fortran 90 notation.

mtype

INTEGER

This scalar value defines the matrix type. Based on this value `pardisoinit` sets default values for the `iparm` array. Refer to the section [Intel MKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel MKL PARDISO

Output Parameters

pt

INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with a dimension of 64. Solver internal data address pointer. These addresses are passed to the solver, and all related internal memory management is organized through this array. The `pardisoinit` routine nullifies the array *pt*.

NOTE

It is very important that the pointer *pt* is initialized with zero before the first call of Intel MKL PARDISO. After that first call you should never modify the pointer, because it could cause a serious memory leak or a crash.

iparm

INTEGER

Array with a dimension of 64. This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver. The `pardisoinit` routine fills-in the `iparm` array with the default values. Refer to the section [Intel MKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel MKL PARDISO.

pardiso_64

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.

Syntax

```
call pardiso_64 (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl, b, x, error)
```

Include Files

- `mkl.fi, mkl_pardiso.f90`

Description

`pardiso_64` is an alternative ILP64 (64-bit integer) version of the `pardiso` routine (see [Description](#) section for more details). The interface of `pardiso_64` is the same as the interface of `pardiso`, but it accepts and returns all INTEGER data as `INTEGER*8`.

Use `pardiso_64` when `pardiso` for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel MKL functionality. In other words, if you use 64-bit integer version (`pardiso_64`), you do not need to re-link your applications with ILP64 libraries. Take into account that `pardiso_64` may perform slower than regular `pardiso` on the reordering and symbolic factorization phase.

NOTE

`pardiso_64` is supported only in the 64-bit libraries. If `pardiso_64` is called from the 32-bit libraries, it returns `error = -12`.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The input parameters of `pardiso_64` are the same as the input parameters of `pardiso`, but `pardiso_64` accepts all INTEGER data as `INTEGER*8`.

Output Parameters

The output parameters of `pardiso_64` are the same as the [output parameters of `pardiso`](#), but `pardiso_64` returns all INTEGER data as `INTEGER*8`.

`pardiso_getenv, pardiso_setenv`

Retrieves additional values from or sets additional values in the Intel MKL PARDISO handle.

Syntax

```
error = pardiso_getenv(handle, param, value)
```

```
error = pardiso_setenv(handle, param, value)
```

Include Files

- `mkl.fi, mkl_pardiso.f90`

NOTE

`pardiso_setenv` requires the `value` parameter to be converted to the string in C notation if it is called from Fortran. You can do this using `mkl_cvt_to_null_terminated_str` subroutine declared in the `mkl_dss.f90` include files (see the example below).

Description

These functions operate with the Intel MKL PARDISO handle. The `pardiso_getenv` routine retrieves additional values from the Intel MKL PARDISO handle, and `pardiso_setenv` sets specified values in the Intel MKL PARDISO handle.

These functions enable retrieving and setting the name of the Intel MKL PARDISO OOC file.

To retrieve the Intel MKL PARDISO OOC file name, you can apply this function to any properly-created handle.

To set the Intel MKL PARDISO OOC file name in the handle you must call the function before the reordering stage. This is because the OOC file name is stored in the handle after the reordering stage and it is not changed during further computations.

NOTE

A 1024-byte internal buffer is used inside Intel MKL PARDISO for storing the OOC file name. Allocate a 1024-byte buffer for passing to the `pardiso_getenv` function as the *value* parameter.

Input Parameters

<i>handle</i>	Fortran: MKL_FORTRAN_HANDLE Intel MKL PARDISO handle for which to set and from which to retrieve information. (See DSS Interface Description for structure description)
<i>param</i>	Fortran: INTEGER. Specifies the required parameter. The only value is PARDISO_OOC_FILE_NAME, defined in the corresponding include file.
<i>value</i>	Fortran: CHARACTER array. Input parameter for <code>pardiso_setenv</code> . Contains the name of the OOC file that must be used in the handle.

Output Parameters

<i>handle</i>	Output parameter for <code>pardiso_setenv</code> . Data object of the MKL_DSS_HANDLE type (see DSS Interface Description).
<i>value</i>	Output parameter for <code>pardiso_getenv</code> . Contains the name of the OOC file that must be used in the handle.

Example (FORTRAN 90)

```

INCLUDE 'mkl_pardiso.f90'
INCLUDE 'mkl_dss.f90'
PROGRAM pardiso_sym_f90
USE mkl_pardiso
USE mkl_dss

INTEGER*8 pt(64)
CHARACTER*1024 file_name
INTEGER buff(256), bufLen, error

pt(1:64) = 0

file_name = 'pardiso_ooc_file'
bufLen = len_trim(file_name)
call mkl_cvt_to_null_terminated_str(buff, bufLen, trim(file_name))
error = pardiso_setenv(pt, PARDISO_OOC_FILE_NAME, buff)

```

```
! call pardiso() here
END PROGRAM
```

mkl_pardiso_pivot

Replaces routine which handles Intel MKL PARDISO pivots with user-defined routine.

Syntax

```
call mkl_pardiso_pivot (ai, bi, eps)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

The `mkl_pardiso_pivot` routine allows you to handle diagonal elements which arise during numerical factorization that are zero or near zero. By default, Intel MKL PARDISO determines that a diagonal element bi is a pivot if $bi < eps$, and if so, replaces it with eps . But you can provide your own routine to modify the resulting factorized matrix in case there are small elements on the diagonal during the factorization step.

NOTE

To use this routine, you must set `iparm(56)` to 1 before the main `pardiso` loop.

NOTE

This routine is only available for in-core Intel MKL PARDISO.

Input Parameters

<i>ai</i>	DOUBLE PRECISION - for real types of matrices (<i>mtype</i> =2, -2, 4, and 6) and for double precision Intel MKL PARDISO (<i>iparm</i> (28)=0) Diagonal element of initial matrix corresponding to pivot element.
<i>bi</i>	DOUBLE PRECISION - for real types of matrices (<i>mtype</i> =2, -2, 4, and 6) and for double precision Intel MKL PARDISO (<i>iparm</i> (28)=0) Diagonal element of factorized matrix that could be chosen as a pivot element.
<i>eps</i>	DOUBLE PRECISION Scalar to compare with diagonal of factorized matrix. On input equal to parameter described by <i>iparm</i> (10).

Output Parameters

<i>bi</i>	In case element is chosen as a pivot, value with which to replace the pivot.
-----------	--

pardiso_getdiag

Returns diagonal elements of initial and factorized matrix.

Syntax

```
call pardiso_getdiag (pt, df, da, mnum, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This routine returns the diagonal elements of the initial and factorized matrix for a real or Hermitian matrix.

NOTE

In order to use this routine, you must set *iparm*(56) to 1 before the main *pardiso* loop.

Input Parameters

<i>pt</i>	INTEGER for 32-bit or 64-bit architectures INTEGER*8 for 64-bit architectures Array with a size of 64. Handle to internal data structure for the Intel MKL PARDISO solver. The entries must be set to zero prior to the first call to <i>pardiso</i> . Unique for factorization.
<i>mnum</i>	INTEGER Indicates the actual matrix for the solution phase of the Intel MKL PARDISO solver. With this scalar you can define the diagonal elements of the factorized matrix that you want to obtain. The value must be: 1 $\leq mnum \leq maxfct$. In most applications this value is 1.

Output Parameters

<i>df</i>	DOUBLE PRECISION - for real types of matrices and for double precision Intel MKL PARDISO (<i>iparm</i> (28)=0) REAL - for real types of matrices and for single precision Intel MKL PARDISO (<i>iparm</i> (28)=1) DOUBLE COMPLEX - for complex types of matrices and for double precision Intel MKL PARDISO (<i>iparm</i> (28)=0) COMPLEX - for complex types of matrices and for single precision Intel MKL PARDISO (<i>iparm</i> (28)=1) Array with a dimension of <i>n</i> . Contains diagonal elements of the factorized matrix after factorization.
-----------	--

NOTE

Elements of *df* correspond to diagonal elements of matrix *L* computed during phase 22. Because during phase 22 Intel MKL PARDISO makes additional permutations to improve stability, it is possible that array *df* is not in line with the *perm* array computed during phase 11.

da DOUBLE PRECISION - for real types of matrices and for double precision Intel MKL PARDISO (*iparm*(28)=0)
 REAL - for real types of matrices and for single precision Intel MKL PARDISO (*iparm*(28)=1)
 DOUBLE COMPLEX - for complex types of matrices and for double precision Intel MKL PARDISO (*iparm*(28)=0)
 COMPLEX - for complex types of matrices and for single precision Intel MKL PARDISO (*iparm*(28)=1)
 Array with a dimension of *n*. Contains diagonal elements of the initial matrix.

NOTE

Elements of *da* correspond to diagonal elements of matrix *L* computed during phase 22. Because during phase 22 Intel MKL PARDISO makes additional permutations to improve stability, it is possible that array *da* is not in line with the *perm* array computed during phase 11.

error INTEGER
 The error indicator.

<i>error</i>	Information
0	no error
-1	Diagonal information not turned on before <i>pardiso</i> main loop (<i>iparm</i> (56)=0).

pardiso_handle_store

Store internal structures from pardiso to a file.

Syntax

call *pardiso_handle_store* (*pt*, *dirname*, *error*)

Include Files

- *mkl.fi*, *mkl_pardiso.f90*

Description

This function stores Intel MKL PARDISO structures to a file, allowing you to store Intel MKL PARDISO internal structures between the stages of the *pardiso* routine. The *pardiso_handle_restore* routine can restore the Intel MKL PARDISO internal structures from the file.

Input Parameters

<i>pt</i>	<p>INTEGER for 32-bit or 64-bit architectures</p> <p>INTEGER*8 for 64-bit architectures</p> <p>Array with a size of 64. Handle to internal data structure.</p>
<i>dirname</i>	<p>CHARACTER</p> <p>String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named <code>handle.pds</code> in the directory.</p>

Output Parameters

<i>pt</i>	Handle to internal data structure.												
<i>error</i>	<p>INTEGER</p> <p>The error indicator.</p> <table> <thead> <tr> <th>error</th> <th>Information</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>No error.</td> </tr> <tr> <td>-2</td> <td>Not enough memory.</td> </tr> <tr> <td>-10</td> <td>Cannot open file for writing.</td> </tr> <tr> <td>-11</td> <td>Error while writing to file.</td> </tr> <tr> <td>-13</td> <td>Wrong file format.</td> </tr> </tbody> </table>	error	Information	0	No error.	-2	Not enough memory.	-10	Cannot open file for writing.	-11	Error while writing to file.	-13	Wrong file format.
error	Information												
0	No error.												
-2	Not enough memory.												
-10	Cannot open file for writing.												
-11	Error while writing to file.												
-13	Wrong file format.												

pardiso_handle_restore

Restore pardiso internal structures from a file.

Syntax

```
call pardiso_handle_restore (pt, dirname, error)
```

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function restores Intel MKL PARDISO structures from a file. This allows you to restore Intel MKL PARDISO internal structures stored by `pardiso_handle_store` after a phase of the `pardiso` routine and continue execution of the next phase.

Input Parameters

<i>dirname</i>	<p>CHARACTER</p> <p>String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.</p>
----------------	---

Output Parameters

pt INTEGER for 32-bit or 64-bit architectures
 INTEGER*8 for 64-bit architectures
 Array with a dimension of 64. Handle to internal data structure.

error INTEGER
 The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete

Delete files with pardiso internal structure data.

Syntax

call `pardiso_handle_delete (dirname, error)`

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function deletes files generated with `pardiso_handle_store` that contain Intel MKL PARDISO internal structures.

Input Parameters

dirname CHARACTER
 String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error INTEGER
 The error indicator.

error	Information
0	No error.
-10	Cannot delete files.

pardiso_handle_store_64

Store internal structures from `pardiso_64` to a file.

Syntax

call `pardiso_handle_store_64 (pt, dirname, error)`

Include Files

- `mkl.fi`, `mkl_pardiso.f90`

Description

This function stores Intel MKL PARDISO structures to a file, allowing you to store Intel MKL PARDISO internal structures between the stages of the `pardiso_64` routine. The `pardiso_handle_restore_64` routine can restore the Intel MKL PARDISO internal structures from the file.

Input Parameters

<i>pt</i>	INTEGER*8 for 64-bit architectures Array with a dimension of 64. Handle to internal data structure.
<i>dirname</i>	CHARACTER String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named <code>handle.pds</code> in the directory.

Output Parameters

<i>pt</i>	Handle to internal data structure.
<i>error</i>	INTEGER The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for writing.
-11	Error while writing to file.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.
-13	Wrong file format.

pardiso_handle_restore_64

Restore `pardiso_64` internal structures from a file.

Syntax

```
call pardiso_handle_restore_64 (pt, dirname, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This function restores Intel MKL PARDISO structures from a file. This allows you to restore Intel MKL PARDISO internal structures stored by `pardiso_handle_store_64` after a phase of the `pardiso_64` routine and continue execution of the next phase.

Input Parameters

<i>dirname</i>	CHARACTER
	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Input Parameters

<i>pt</i>	INTEGER for 32-bit or 64-bit architectures INTEGER*8 for 64-bit architectures Array with a dimension of 64. Handle to internal data structure.
-----------	--

<i>error</i>	INTEGER The error indicator.
--------------	---------------------------------

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete_64

Syntax

Delete files with `pardiso_64` internal structure data.

```
call pardiso_handle_delete_64 (dirname, error)
```

Include Files

- mkl.fi, mkl_pardiso.f90

Description

This function deletes files generated with `pardiso_handle_store_64` that contain Intel MKL PARDISO internal structures.

Input Parameters

dirname CHARACTER
String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error INTEGER
The error indicator.

error	Information
0	No error.
-10	Cannot delete files.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.

Intel MKL PARDISO Parameters in Tabular Form

The following table lists all parameters of Intel MKL PARDISO and gives their brief descriptions.

Parameter	Type	Description	Values	Comments	In/Out
<i>pt</i> (64)	FORTRAN 77: INTEGER on 32-bit architectures, INTEGER*8 on 64-bit architectures Fortran 90: TYPE (MKL_PARDISO_HANDLE), INTENT (INOUT)	Solver internal data address pointer	0	Must be initialized with zeros and never be modified later	in/out
<i>maxfct</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)	Maximal number of factors in memory	>0	Generally used value is 1	in
<i>mnum</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)	The number of matrix (from 1 to <i>maxfct</i>) to solve	[1: <i>maxfct</i>]	Generally used value is 1	in
<i>mtype</i>	FORTRAN 77: INTEGER	Matrix type	1	Real and structurally symmetric	in

Parameter	Type	Description	Values	Comments	In/Out
	Fortran 90: INTEGER, INTENT (IN)		2	Real and symmetric positive definite	
			-2	Real and symmetric indefinite	
			3	Complex and structurally symmetric	
			4	Complex and Hermitian positive definite	
			-4	Complex and Hermitian indefinite	
			6	Complex and symmetric matrix	
			11	Real and nonsymmetric matrix	
			13	Complex and nonsymmetric matrix	
<i>phase</i>	FORTTRAN 77: INTEGER	Controls the execution of the solver	11	Analysis	in
	Fortran 90: INTEGER, INTENT (IN)	For <i>iparm</i> (36) > 0, phases 331, 332, and 333 perform a different decomposition. See the <i>phase</i> parameter of pardiso for details.	12	Analysis, numerical factorization	
			13	Analysis, numerical factorization, solve	
			22	Numerical factorization	
			23	Numerical factorization, solve	
			33	Solve, iterative refinement	
			331	<i>phase</i> =33, but only forward substitution	
			332	<i>phase</i> =33, but only diagonal substitution	
			333	<i>phase</i> =33, but only backward substitution	
			0	Release internal memory for L and U of the matrix number <i>mnum</i>	
			-1	Release all internal memory for all matrices	
<i>n</i>	FORTTRAN 77: INTEGER	Number of equations in the sparse linear system $A*X = B$	>0		in
	Fortran 90: INTEGER, INTENT (IN)				

Parameter	Type	Description	Values	Comments	In/Out
$a(*)$	FORTRAN 77: PARDISO_DATA_TY PE ¹) Fortran 90: PARDISO_DATA_TY PE ¹), INTENT (IN)	Contains the non-zero elements of the coefficient matrix A	*	The size of a is the same as that of ja , and the coefficient matrix can be either real or complex. The matrix must be stored in the 3-array variation of compressed sparse row (CSR3) format with increasing values of ja for each row	in
$ia(n + 1)$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)	<i>rowIndex</i> array in CSR3 format	≥ 0	$ia(i)$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia(n+1)$ (for one-based indexing) or $ia(n)$ (for zero-based indexing) is taken to be equal to the number of non-zero elements in A , plus one. Note: $iparm(35)$ indicates whether row/column indexing starts from 1 or 0.	in
$ja(*)$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)	<i>columns</i> array in CSR3 format	≥ 0	The column indices for each row of A must be sorted in increasing order. For structurally symmetric matrices zero diagonal elements must be stored in a and ja . Zero diagonal elements should be stored for symmetric matrices, although they are not required. For symmetric matrices, the solver needs only the upper triangular part of the system. Note: $iparm(35)$ indicates whether row/column indexing starts from 1 or 0.	in
$perm(n)$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (INOUT)	Holds the permutation vector of size n or specifies elements used for computing a partial solution	≥ 0	You can apply your own fill-in reducing ordering ($iparm(5) = 1$) or return the permutation from the solver ($iparm(5) = 2$). Let $C = P^*A^*P^T$ be the permuted matrix. Row (column) i of C is the $perm(i)$ row (column) of	in/ out

Parameter	Type	Description	Values	Comments	In/Out
				<p>A. The numbering of the array must describe a permutation.</p> <p>To specify elements for a partial solution, set $iparm(5) = 0$, $iparm(31) > 0$, and $iparm(36) \#unique_1077/unique_1077_Connect_42_IPARM36 = 0$.</p> <p>To specify elements for a Schur complement, set $iparm(5) = 0$, $iparm(31) = 0$, and $iparm(36) \#unique_1077/unique_1077_Connect_42_IPARM36 > 0$.</p> <p>Note: $iparm(35)$ indicates whether row/column indexing starts from 1 or 0.</p>	
<i>nrhs</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT (IN)</p>	Number of right-hand sides that need to be solved for	≥ 0	<p>Generally used value is 1</p> <p>To obtain better Intel MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.</p>	in
<i>iparm(64)</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT (INOUT)</p>	This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver (see pardiso iparm Parameter for more details)	*	If $iparm(1)=0$, Intel MKL PARDISO fills $iparm(2)$ through $iparm(64)$ with default values and uses them.	in/out
<i>msglvl</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT (IN)</p>	Message level information	0 1	<p>Intel MKL PARDISO generates no output</p> <p>Intel MKL PARDISO prints statistical information</p>	in
<i>b(n*nrhs)</i>	FORTRAN 77: PARDISO_DATA_TY PE ¹)	Right-hand side vectors	*	On entry, contains the right-hand side vector/matrix <i>B</i> , which is placed contiguously in memory. The $b(i+(k-1) \times nrhs)$ element must hold the <i>i</i> -	in/out

Parameter	Type	Description	Values	Comments	In/Out
	Fortran 90: PARDISO_DATA_TY PE ¹ , INTENT (INOUT)			th component of k -th right-hand side vector. Note that b is only accessed in the solution phase. On output, the array is replaced with the solution if $iparm(6) = 1$.	
$x(n*nrhs)$	FORTTRAN 77: PARDISO_DATA_TY PE ¹ Fortran 90: PARDISO_DATA_TY PE ¹ , INTENT (OUT)	Solution vectors	*	On output, if $iparm(6) = 0$, contains solution vector/matrix X which is placed contiguously in memory. The $x(i+(k-1)*nrhs)$ element must hold the i -th component of k -th solution vector. Note that x is only accessed in the solution phase.	out
$error$	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (OUT)	Error indicator	0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11	No error Input inconsistent Not enough memory Reordering problem Zero pivot, numerical factorization or iterative refinement problem Unclassified (internal) error Reordering failed (matrix types 11 and 13 only) Diagonal matrix is singular 32-bit integer overflow problem Not enough memory for OOC Problems with opening OOC temporary files Read/write problems with the OOC data file	out

¹) See description of PARDISO_DATA_TYPE in [PARDISO_DATA_TYPE](#).

pardiso iparm Parameter

The following table describes all individual components of the Intel MKL PARDISO $iparm$ parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description
<i>iparm</i> (1)	Use default values.
input	<p>0 <i>iparm</i>(2) - <i>iparm</i>(64) are filled with default values.</p> <p>!=0 You must supply all values in components <i>iparm</i>(2) - <i>iparm</i>(64).</p>
<i>iparm</i> (2)	Fill-in reducing ordering for the input matrix.
input	<p>CAUTION You can control the parallel execution of the solver by explicitly setting the <code>MKL_NUM_THREADS</code> environment variable. If fewer OpenMP threads are available than specified, the execution may slow down instead of speeding up. If <code>MKL_NUM_THREADS</code> is not defined, then the solver uses all available processors.</p> <hr/> <p>0 The minimum degree algorithm [Li99].</p> <p>2* The nested dissection algorithm from the METIS package [Karypis98].</p> <p>3 The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel MKL PARDISO Phase 1 takes significant time.</p> <hr/> <p>NOTE Setting <i>iparm</i>(2) = 3 prevents the use of CNR mode (<i>iparm</i>(34) > 0) because Intel MKL PARDISO uses dynamic parallelism.</p>
<i>iparm</i> (3)	Reserved. Set to zero.
<i>iparm</i> (4)	Preconditioned CGS/CG.
input	<p>This parameter controls preconditioned CGS [Sonn89] for nonsymmetric or structurally symmetric matrices and Conjugate-Gradients for symmetric matrices. <i>iparm</i>(4) has the form $iparm(4) = 10 * L + K$.</p> <hr/> <p>K=0 The factorization is always computed as required by <i>phase</i>.</p> <hr/> <p>K=1 CGS iteration replaces the computation of <i>LU</i>. The preconditioner is <i>LU</i> that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</p> <hr/> <p>K=2 CGS iteration for symmetric positive definite matrices replaces the computation of LL^T. The preconditioner is LL^T that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</p> <hr/> <p>The value <i>L</i> controls the stopping criterion of the Krylow-Subspace iteration: $eps_{CGS} = 10^{-L}$ is used in the stopping criterion $dx_i / dx_0 < eps_{CGS}$ where $dx_i = inv(L*U)*r_i$ for <i>K</i> = 1 or $dx_i = inv(L*L^T)*r_i$ for <i>K</i> = 2 and <i>r_i</i> is the residue at iteration <i>i</i> of the preconditioned Krylow-Subspace iteration.</p> <p>A maximum number of 150 iterations is fixed with the assumption that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residue excursions are checked and can terminate the iteration process. If <i>phase</i> =23, then the factorization for a given <i>A</i> is automatically recomputed in cases where the Krylow-Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylow-Subspace iteration is returned. Using <i>phase</i> =33 results in an error message (<i>error</i>=-4) if the stopping criteria for the Krylow-Subspace iteration can not be reached. More information on the failure can be obtained from <i>iparm</i>(20).</p>

Component	Description								
	<p>The default is $iparm(4)=0$, and other values are only recommended for an advanced user. $iparm(4)$ must be greater than or equal to zero.</p> <p>Examples:</p> <table border="1"> <thead> <tr> <th>$iparm(4)$</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>31</td> <td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices</td> </tr> <tr> <td>61</td> <td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices</td> </tr> <tr> <td>62</td> <td>LL^T-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices</td> </tr> </tbody> </table>	$iparm(4)$	Description	31	LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices	61	LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices	62	LL^T -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices
$iparm(4)$	Description								
31	LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices								
61	LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices								
62	LL^T -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices								
$iparm(5)$ input	<p>User permutation.</p> <p>This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel MKL PARDISO.</p> <p>This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^*A^*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the $perm$ parameter.</p> <hr/> <p>CAUTION You can only set one of $iparm(5)$, $iparm(31)$, and $iparm(36)$, so be sure that the $iparm(31)$ (partial solution) and the $iparm(36)$ (Schur complement) parameters are 0 if you set $iparm(5)$.</p> <hr/> <p>0* User permutation in the $perm$ array is ignored.</p> <hr/> <p>1 Intel MKL PARDISO uses the user supplied fill-in reducing permutation from the $perm$ array. $iparm(2)$ is ignored.</p> <hr/> <p>NOTE Setting $iparm(5) = 1$ prevents use of a parallel algorithm for the solve step.</p> <hr/> <p>2 Intel MKL PARDISO returns the permutation vector computed at phase 1 in the $perm$ array.</p>								
$iparm(6)$ input	<p>Write solution on x.</p> <hr/> <p>NOTE The array x is always used.</p> <hr/> <p>0* The array x contains the solution; right-hand side vector b is kept unchanged.</p> <hr/> <p>1 The solver stores the solution on the right-hand side b.</p>								
$iparm(7)$ output	<p>Number of iterative refinement steps performed.</p> <p>Reports the number of iterative refinement steps that were actually performed during the solve step.</p>								
$iparm(8)$	Iterative refinement step.								

Component	Description
input	On entry to the solve and iterative refinement step, $iparm(8)$ must be set to the maximum number of iterative refinement steps that the solver performs.
	0* The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.
	>0 Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of $iparm(8)$ steps of iterative refinement. The solver might stop the process before the maximum number of steps if <ul style="list-style-type: none"> • a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case Intel MKL PARDISO returns -4 in the <i>error</i> parameter. The number of executed iterations is reported in $iparm(7)$.
	<0 Same as above, but the accumulation of the residuum uses extended precision real and complex data types. <p>Perturbed pivots result in iterative refinement (independent of $iparm(8)=0$) and the number of executed iterations is reported in $iparm(7)$.</p>
$iparm(9)$	Reserved. Set to zero.
$iparm(10)$	Pivoting perturbation.
input	<p>This parameter instructs Intel MKL PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices ($mtype = 11$ or $mtype = 13$) and symmetric matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].</p> <p>Small pivots are perturbed with $eps = 10^{-iparm(10)}$.</p> <p>The magnitude of the potential pivot is tested against a constant threshold of $alpha = eps * A2 _{inf}$, where $eps = 10^{-iparm(10)}$, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $A2 _{inf}$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the sign $(I_{II}) * eps * A2 _{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $eps = 10^{-iparm(10)}$.</p>
	13* The default value for nonsymmetric matrices ($mtype = 11$, $mtype = 13$), $eps = 10^{-13}$.
	8* The default value for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, $mtype = 6$), $eps = 10^{-8}$.
$iparm(11)$	Scaling vectors.
input	Intel MKL PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less than or equal to 1. This scaling method is applied only to nonsymmetric matrices ($mtype = 11$ or $mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$) when the symmetric weighted matchings are applied ($iparm(13) = 1$).

Component	Description
	Use <code>iparm(11) = 1</code> (scaling) and <code>iparm(13) = 1</code> (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (<code>phase=11</code>) you must provide the numerical values of the matrix <i>A</i> in array <i>a</i> in case of scaling and symmetric weighted matching.
	0* Disable scaling. Default for symmetric indefinite matrices.
	1* Enable scaling. Default for nonsymmetric matrices. Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices (<code>mtype = 11</code> , <code>mtype = 13</code>). The scaling can also be used for symmetric indefinite matrices (<code>mtype = -2</code> , <code>mtype = -4</code> , <code>mtype = 6</code>) when the symmetric weighted matchings are applied (<code>iparm(13) = 1</code>). Note that in the analysis phase (<code>phase=11</code>) you must provide the numerical values of the matrix <i>A</i> in case of scaling.
<code>iparm(12)</code> input	Solve with transposed or conjugate transposed matrix <i>A</i> .
	NOTE For real matrices the terms <i>transposed</i> and <i>conjugate transposed</i> are equivalent.
	0* Solve a linear system $AX = B$.
	1 Solve a conjugate transposed system $A^H X = B$ based on the factorization of the matrix <i>A</i> .
	2 Solve a transposed system $A^T X = B$ based on the factorization of the matrix <i>A</i> .
<code>iparm(13)</code> input	Improved accuracy using (non-) symmetric weighted matching. Intel MKL PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.
	0* Disable matching. Default for symmetric indefinite matrices.
	1* Enable matching. Default for nonsymmetric matrices. Maximum weighted matching algorithm to permute large elements close to the diagonal. It is recommended to use <code>iparm(11) = 1</code> (scaling) and <code>iparm(13) = 1</code> (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems. Note that in the analysis phase (<code>phase=11</code>) you must provide the numerical values of the matrix <i>A</i> in case of symmetric weighted matching.
<code>iparm(14)</code> output	Number of perturbed pivots. After factorization, contains the number of perturbed pivots for the matrix types: 11, 13, -2, -4 and -6.
<code>iparm(15)</code> output	Peak memory on symbolic factorization. The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.

Component	Description				
<i>iparm</i> (16) output	Permanent memory on symbolic factorization. Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases. This value is only computed in phase 1.				
<i>iparm</i> (17) output	Size of factors/Peak memory on numerical factorization and solution. This parameter provides the size in kilobytes of the total memory consumed by in-core Intel MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. See <i>iparm</i> (63) for the OOC mode. The total peak memory consumed by Intel MKL PARDISO is $\max(iparm(15), iparm(16)+iparm(17))$				
<i>iparm</i> (18) input/output	Report the number of non-zero elements in the factors. <table border="1"> <tr> <td><0</td> <td>Enable reporting if <i>iparm</i>(18) < 0 on entry. The default value is -1.</td> </tr> <tr> <td>>=0</td> <td>Disable reporting.</td> </tr> </table>	<0	Enable reporting if <i>iparm</i> (18) < 0 on entry. The default value is -1.	>=0	Disable reporting.
<0	Enable reporting if <i>iparm</i> (18) < 0 on entry. The default value is -1.				
>=0	Disable reporting.				
<i>iparm</i> (19) input/output	Report number of floating point operations (in 10 ⁶ floating point operations) that are necessary to factor the matrix <i>A</i> . <table border="1"> <tr> <td><0</td> <td>Enable report if <i>iparm</i>(19) < 0 on entry. This increases the reordering time.</td> </tr> <tr> <td>>=0</td> <td>Disable report.</td> </tr> </table>	<0	Enable report if <i>iparm</i> (19) < 0 on entry. This increases the reordering time.	>=0	Disable report.
<0	Enable report if <i>iparm</i> (19) < 0 on entry. This increases the reordering time.				
>=0	Disable report.				
<i>iparm</i> (20) output	Report CG/CGS diagnostics. <table border="1"> <tr> <td>>0</td> <td>CGS succeeded, reports the number of completed iterations.</td> </tr> <tr> <td><0</td> <td>CG/CGS failed (<i>error</i>=-4 after the solution phase). If <i>phase</i>= 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i>=0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i>(20)= - <i>it_cgs</i>*10 - <i>cgs_error</i>. Possible values of <i>cgs_error</i>: 1 - fluctuations of the residuum are too large 2 - $\ dx_{\max_it_cgs}/2\$ is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement 5 - factorization is too fast for this matrix. It is better to use the factorization method with <i>iparm</i>(4) = 0</td> </tr> </table>	>0	CGS succeeded, reports the number of completed iterations.	<0	CG/CGS failed (<i>error</i> =-4 after the solution phase). If <i>phase</i> = 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i> =0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i> (20)= - <i>it_cgs</i> *10 - <i>cgs_error</i> . Possible values of <i>cgs_error</i> : 1 - fluctuations of the residuum are too large 2 - $\ dx_{\max_it_cgs}/2\ $ is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement 5 - factorization is too fast for this matrix. It is better to use the factorization method with <i>iparm</i> (4) = 0
>0	CGS succeeded, reports the number of completed iterations.				
<0	CG/CGS failed (<i>error</i> =-4 after the solution phase). If <i>phase</i> = 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i> =0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i> (20)= - <i>it_cgs</i> *10 - <i>cgs_error</i> . Possible values of <i>cgs_error</i> : 1 - fluctuations of the residuum are too large 2 - $\ dx_{\max_it_cgs}/2\ $ is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement 5 - factorization is too fast for this matrix. It is better to use the factorization method with <i>iparm</i> (4) = 0				
<i>iparm</i> (21) input	Pivoting for symmetric indefinite matrices. NOTE Use <i>iparm</i> (11) = 1 (scaling) and <i>iparm</i> (13) = 1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.				
	0 Apply 1x1 diagonal pivoting during the factorization process.				

Component	Description
	<p>1* Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <i>mtype</i>=-2, <i>mtype</i>=-4, or <i>mtype</i>=6.</p>
	<p>2 Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using <i>iparm</i>(21) = 0 except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <i>iparm</i>(8) (0 by default).</p>
	<p>3 Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <i>mtype</i>=-2, <i>mtype</i>=-4, or <i>mtype</i>=6. Using this value is the same as using <i>iparm</i>(21) = 1 except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <i>iparm</i>(8) (0 by default).</p>
<i>iparm</i> (22) output	<p>Inertia: number of positive eigenvalues. Intel MKL PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.</p>
<i>iparm</i> (23) output	<p>Inertia: number of negative eigenvalues. Intel MKL PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.</p>
<i>iparm</i> (24) input	<p>Parallel factorization control.</p> <hr/> <p>NOTE The two-level factorization algorithm does not improve performance in OOC mode.</p> <hr/>
	<p>0* Intel MKL PARDISO uses the classic algorithm for factorization.</p>
	<p>1 Intel MKL PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight).</p>
<i>iparm</i> (25) input	<p>Parallel forward/backward solve control.</p> <hr/> <p>0* Intel MKL PARDISO uses a parallel algorithm for the solve step.</p> <hr/> <p>1 Intel MKL PARDISO uses the sequential forward and backward solve. This feature is available only for in-core Intel MKL PARDISO (see iparm(60)).</p>
<i>iparm</i> (26)	<p>Reserved. Set to zero.</p>
<i>iparm</i> (27) input	<p>Matrix checker.</p> <hr/> <p>0* Intel MKL PARDISO does not check the sparse matrix representation for errors.</p> <hr/> <p>1 Intel MKL PARDISO checks integer arrays <i>ia</i> and <i>ja</i>. In particular, Intel MKL PARDISO checks whether column indices are sorted in increasing order within each row.</p>
<i>iparm</i> (28) input	<p>Single or double precision Intel MKL PARDISO. See iparm(8) for information on controlling the precision of the refinement steps.</p>

Component	Description
	<p>Important The <i>iparm(28)</i> value is stored in the Intel MKL PARDISO handle between Intel MKL PARDISO calls, so the precision mode can be changed only during phase 1.</p>
	<p>0* Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.</p>
	<p>1 Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.</p>
<i>iparm(29)</i>	Reserved. Set to zero.
<i>iparm(30)</i> output	<p>Number of zero or negative pivots. If Intel MKL PARDISO detects zero or negative pivot for <i>mtype=2</i> or <i>mtype=4</i> matrix types, the factorization is stopped, Intel MKL PARDISO returns immediately with an <i>error</i> = -4, and <i>iparm(30)</i> reports the number of the equation where the first zero or negative pivot is detected.</p>
<i>iparm(31)</i> input	<p>Partial solve and computing selected components of the solution vectors. This parameter controls the solve step of Intel MKL PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define <i>perm</i> so that when $perm(i) = 1$ it means that either the <i>i</i>-th component in the right-hand sides is nonzero, or the <i>i</i>-th component in the solution vectors is computed, or both, depending on the value of <i>iparm(31)</i>. The permutation vector <i>perm</i> must be present in all phases of Intel MKL PARDISO software. At the reordering step, the software overwrites the input vector <i>perm</i> by a permutation vector used by the software at the factorization and solver step. If <i>m</i> is the number of components such that $perm(i) = 1$, then the last <i>m</i> components of the output vector <i>perm</i> are a set of the indices <i>i</i> satisfying the condition $perm(i) = 1$ on input.</p>
	<p>NOTE Turning on this option often increases the time used by Intel MKL PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.</p>
	<p>Important This feature is only available for in-core Intel MKL PARDISO, so to use it you must set <i>iparm(60)</i> = 0. Set the parameters <i>iparm(8)</i> (iterative refinement steps), <i>iparm(4)</i> (preconditioned CGS), <i>iparm(5)</i> (user permutation), and <i>iparm(36)</i> (Schur complement) to 0 as well.</p>
	<p>0* Disables this option.</p>
	<p>1 it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the (<i>i</i>)-th component in the right-hand sides is nonzero. In this case Intel MKL PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the <i>i</i>-th component in the solution vectors is only computed if $perm(i) = 1$.</p>

Component	Description
2	<p>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the <i>i</i>-th component in the right-hand sides is nonzero.</p> <p>Unlike for $iparm(31) = 1$, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for $iparm(31) = 2$ you must set the <i>i</i>-th component of the right-hand sides to zero explicitly if $perm(i)$ is not equal to 1.</p>
3	<p>Selected components of the solution vectors are computed. The <i>perm</i> array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case $perm(i) = 1$ means that the <i>i</i>-th component in the solution vectors is computed.</p>
<i>iparm</i> (32)	Reserved. Set to zero.
<i>iparm</i> (33)	
<i>iparm</i> (34)	<p>Optimal number of OpenMP threads for conditional numerical reproducibility (CNR) mode. Intel MKL PARDISO reads the value of <i>iparm</i>(34) during the analysis phase (phase 1), so you cannot change it later.</p> <p>Because Intel MKL PARDISO uses C random number generator facilities during the analysis phase (phase 1) you must take these precautions to get numerically reproducible results:</p> <ul style="list-style-type: none"> • Do not alter the states of the random number generators. • Do not run multiple instances of Intel MKL PARDISO in parallel in the analysis phase (phase 1).
input	
	<p>NOTE</p> <p>CNR is only available for the in-core version of Intel MKL PARDISO and the non-parallel version of the nested dissection algorithm. You must also:</p> <ul style="list-style-type: none"> • set <i>iparm</i>(60) to 0 in order to use the in-core version, • not set <i>iparm</i>(2) to 3 in order to not use the parallel version of the nested dissection algorithm. <p>Otherwise Intel MKL PARDISO does not produce numerically repeatable results even if CNR is enabled for Intel MKL using the functionality described in Support Functions for CNR.</p>
0*	<p>CNR mode for Intel MKL PARDISO is enabled only if it is enabled for Intel MKL using the functionality described in Support Functions for CNR and the in-core version is used. Intel MKL PARDISO determines the optimal number of OpenMP threads automatically, and produces numerically reproducible results regardless of the number of threads.</p>
>0	<p>CNR mode is enabled for Intel MKL PARDISO if in-core version is used and the optimal number of OpenMP threads for Intel MKL PARDISO to rely on is defined by the value of <i>iparm</i>(34). You can use <i>iparm</i>(34) to enable CNR mode independent from other Intel MKL domains. To get the best performance, set <i>iparm</i>(34) to the actual number of hardware threads dedicated for Intel MKL PARDISO. Setting <i>iparm</i>(34) to fewer OpenMP threads than the maximum number of them in use reduces the scalability of the problem being solved. Setting <i>iparm</i>(34) to more threads than are available can reduce the performance of Intel MKL PARDISO.</p>
<i>iparm</i> (35)	One- or zero-based indexing of columns and rows.
input	<p>0*</p> <p>One-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 1 (Fortran-style indexing).</p>

Component	Description
	1 Zero-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 0 (C-style indexing).
<i>iparm</i> (36) input	Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector <i>perm</i> to a set of indexes such that when $perm(i) = 1$, the <i>i</i> -th element of the initial matrix is an element of the Schur matrix.
	<p>CAUTION You can only set one of <i>iparm</i>(5), <i>iparm</i>(31), and <i>iparm</i>(36), so be sure that the <i>iparm</i>(5) (user permutation) and the <i>iparm</i>(31) (partial solution) parameters are 0 if you set <i>iparm</i>(36).</p>
	0* Do not compute Schur complement.
	1 Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector.
	<p>NOTE This option only computes the Schur complement matrix, and does not calculate factorization arrays.</p>
	2 Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.
<i>iparm</i> (37) input	Format for matrix storage.
	0* Use CSR format (see Three Array Variation of BSR Format) for matrix storage.
	> 0 Use BSR format (see Three Array Variation of BSR Format) for matrix storage with blocks of size <i>iparm</i> (37).
	<p>NOTE Intel MKL does not support BSR format in these cases:</p> <ul style="list-style-type: none"> <i>iparm</i>(11) > 0 Scaling vectors <i>iparm</i>(13) > 0 Weighted matching <i>iparm</i>(31) > 0 Partial solution <i>iparm</i>(36) > 0 Schur complement <i>iparm</i>(56) > 0 Pivoting control <i>iparm</i>(60) > 0 OOC Intel MKL PARDISO
<i>iparm</i> (38) -	Reserved. Set to zero.
<i>iparm</i> (55) <i>iparm</i> (56)	Diagonal and pivoting control.
	0* Internal function used to work with pivot and calculation of diagonal arrays turned off.

Component	Description
	<p>1 You can use the <code>mkl_pardiso_pivot</code> callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the <code>pardiso</code> factorization step diagonal using the <code>pardiso_getdiag</code> routine. This parameter can be turned on only in the in-core version of Intel MKL PARDISO.</p>
<code>iparm(57)</code>	Reserved. Set to zero.
<code>iparm(59)</code>	
<code>iparm(60)</code>	Intel MKL PARDISO mode.
input	<p><code>iparm(60)</code> switches between in-core (IC) and out-of-core (OOC) Intel MKL PARDISO. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC.</p> <p>Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better Intel MKL PARDISO performance by setting <code>iparm(60)</code> before the reordering phase.</p>
	<p>NOTE The amount of memory used in OOC mode depends on the number of OpenMP threads.</p>
	<p>WARNING Do not increase the number of OpenMP threads used for Intel MKL PARDISO between the first call to <code>pardiso</code> and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.</p>
0*	IC mode.
1	<p>IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> (default value 2000 MB) and <code>MKL_PARDISO_OOC_MAX_SWAP_SIZE</code> (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>.</p> <p>If the total peak memory needed for storing the local arrays is more than <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>, increase <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> if possible.</p>
	<p>NOTE Conditional numerical reproducibility (CNR) is not supported for this mode.</p>
2	<p>OOC mode.</p> <p>The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode.</p> <p>If the total peak memory needed for storing the local arrays is more than <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>, increase <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> if possible.</p>

Component	Description
	To obtain better Intel MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.
<i>iparm</i> (61)	Reserved. Set to zero.
<i>iparm</i> (62)	
<i>iparm</i> (63)	Size of the minimum OOC memory for numerical factorization and solution.
output	This parameter provides the size in kilobytes of the minimum memory required by OOC Intel MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel MKL PARDISO can be estimated as $\max(iparm(15), iparm(16) + iparm(63))$.
<i>iparm</i> (64)	Reserved. Set to zero.

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

PARDISO_DATA_TYPE

The following table lists the values of `PARDISO_DATA_TYPE` depending on the matrix types and values of the parameter `iparm(28)`.

Data type value	Matrix type <i>mtype</i>	<i>iparm</i> (28)	comments
DOUBLE PRECISION	1, 2, -2, 11	0	Real matrices, double precision
REAL		1	Real matrices, single precision
DOUBLE COMPLEX	3, 6, 13, 4, -4	0	Complex matrices, double precision
COMPLEX		1	Complex matrices, single precision

Parallel Direct Sparse Solver for Clusters Interface

The Parallel Direct Sparse Solver for Clusters Interface solves large linear systems of equations with sparse matrices on clusters. It is

- high performing
- robust
- memory efficient
- easy to use

A hybrid implementation combines Message Passing Interface (MPI) technology for data exchange between parallel tasks (processes) running on different nodes, and OpenMP* technology for parallelism inside each node of the cluster. This approach effectively uses modern hardware resources such as clusters consisting of nodes with multi-core processors. The solver code is optimized for the latest Intel processors, but also performs well on clusters consisting of non-Intel processors.

Code examples are available in the Intel MKL installation `examples` directory.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Parallel Direct Sparse Solver for Clusters Interface Algorithm

Parallel Direct Sparse Solver for Clusters Interface solves a set of sparse linear equations

$$A * X = B$$

with multiple right-hand sides using a distributed LU , LL^T , LDL^T or LDL^* factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ matrices.

The solution comprises four tasks:

- analysis and symbolic factorization;
- numerical factorization;
- forward and backward substitution including iterative refinement;
- termination to release all internal solver memory.

The solver first computes a symmetric fill-in reducing permutation P based on the nested dissection algorithm from the METIS package [Karypis98] (included with Intel MKL), followed by the Cholesky or other type of factorization (depending on matrix type) [Schenk00-2] of PAP^T . The solver uses either diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite or Hermitian matrices before finding an approximation of X by forward and backward substitution and iterative refinement.

The initial matrix A is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal blocks. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricted notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. For a large set of matrices from different application areas, the accuracy of this method is comparable to a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Parallel Direct Sparse Solver for Clusters additionally improves the pivoting accuracy by applying symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix A that, if permuted close to the diagonal, enable the factorization process to identify more acceptable pivots and

proceed with fewer pivot perturbations. The methods are based on maximum weighted matching and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

Parallel Direct Sparse Solver for Clusters Interface Matrix Storage

The sparse data storage in the Parallel Direct Sparse Solver for Clusters Interface follows the scheme described in the [Sparse Matrix Storage Formats](#) section using the variable *ja* for *columns*, *ia* for *rowIndex*, and *a* for *values*. Column indices *ja* must be in increasing order per row and each diagonal element to be present in the matrix structure (even if it is zero).

When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a placeholder for that argument.

Algorithm Parallelization and Data Distribution

Intel® MKL Parallel Direct Sparse Solver for Clusters enables parallel execution of the solution algorithm with efficient data distribution.

The master MPI process performs the symbolic factorization phase to represent matrix *A* as computational tree. Then matrix *A* is divided among all MPI processes in a one-dimensional manner. The same distribution is used for *L-factor* (the lower triangular matrix in Cholesky decomposition). Matrix *A* and all required internal data are broadcast to slave MPI processes. Each MPI process fills in its own parts of *L-factor* with initial values of the matrix *A*.

Parallel Direct Sparse Solver for Clusters Interface computes all independent parts of *L-factor* completely in parallel. When a block of the factor must be updated by other blocks, these updates are independently passed to a temporary array on each updating MPI process. It further gathers the result into an updated block using the `MPI_Reduce()` routine. The computations within an MPI process are dynamically divided among OpenMP threads using pipelining parallelism with a combination of left- and right-looking techniques similar to those of the PARDISO* software. Level 3 BLAS operations from Intel MKL ensure highly efficient performance of block-to-block update operations.

During forward/backward substitutions, respective Right Hand Side (RHS) parts are distributed among all MPI processes. All these processes participate in the computation of the solution. Finally, the solution is gathered on the master MPI process.

This approach demonstrates good scalability on clusters with Infiniband* technology. Another advantage of the approach is the effective distribution of *L-factor* among cluster nodes. This enables the solution of tasks with a much higher number of non-zero elements than it is possible with any Symmetric Multiprocessing (SMP) in-core direct solver.

The algorithm ensures that the memory required to keep internal data on each MPI process is decreased when the number of MPI processes in a run increases. However, the solver requires that matrix *A* and some other internal arrays completely fit into the memory of each MPI process.

To get the best performance, run one MPI process per physical node and set the number of OpenMP* threads per node equal to the number of physical cores on the node.

NOTE

Instead of calling `MPI_Init()`, initialize MPI with `MPI_Init_thread()` and set the MPI threading level to `MPI_THREAD_FUNNELED` or higher. For details, see the code examples in `<install_dir>/examples`.

cluster_sparse_solver

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
call cluster_sparse_solver (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs,
iparm, msglvl, b, x, comm, error)
```

Include Files

- Fortran: `mkl_cluster_sparse_solver.f77`
- Fortran 90: `mkl_cluster_sparse_solver.f90`

Description

The routine `cluster_sparse_solver` calculates the solution of a set of sparse linear equations

$$A * X = B$$

with single or multiple right-hand sides, using a parallel *LU*, *LDL*, or *LL^T* factorization, where *A* is an *n*-by-*n* matrix, and *X* and *B* are *n*-by-*nrhs* vectors or matrices.

Input Parameters

NOTE

Most of the input parameters (except for the *pt*, *phase*, and *comm* parameters and, for the distributed format, the *a*, *ia*, and *ja* arrays) must be set on the master MPI process only, and ignored on other processes. Other MPI processes get all required data from the master MPI process using the MPI communicator, *comm*.

pt INTEGER*8 for 64-bit architectures
 Array of size 64.
 Handle to internal data structure. The entries must be set to zero before the first call to `cluster_sparse_solver`.

CAUTION

After the first call to `cluster_sparse_solver` do not modify *pt*, as that could cause a serious memory leak.

maxfct INTEGER
 Ignored; assumed equal to 1.

mnum INTEGER
 Ignored; assumed equal to 1.

mtype INTEGER
 Defines the matrix type, which influences the pivoting method. The Parallel Direct Sparse Solver for Clusters solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric

4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase

INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Parallel Direct Sparse Solver for Clusters has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement
- Memory release (*phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
-1	Release all internal memory for all matrices

n

INTEGER

Number of equations in the sparse linear systems of equations $A^*X = B$.
 Constraint: $n > 0$.

a

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices ($mtype=3, 6, 13, 14$ and -4) and for double precision Parallel Direct Sparse Solver for Clusters Interface ($iparm(28)=0$)

COMPLEX - for complex types of matrices ($mtype=3, 6, 13, 14$ and -4) and for single precision Parallel Direct Sparse Solver for Clusters Interface ($iparm(28)=1$)

Array. Contains the non-zero elements of the coefficient matrix A corresponding to the indices in ja . The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of ja for each row.

For CSR3 format, the size of a is the same as that of ja . Refer to the *values* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format the size of a is the size of ja multiplied by the square of the block size. Refer to the *values* array description in [Three Array Variation of BSR Format](#) for more details.

NOTE

For centralized input ($iparm(40)=0$), provide the a array for the master MPI process only. For distributed assembled input ($iparm(40)=1$ or $iparm(40)=2$), provide it for all MPI processes.

Important

The column indices of non-zero elements of each row of the matrix A must be stored in increasing order, and the diagonal elements must be available and stored in the array, even if they are zero.

ia

INTEGER

For CSR3 format, $ia(i)$ ($i \leq n$) points to the first column index of row i in the array ja . That is, $ia(i)$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zero elements in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format, $ia(i)$ ($i \leq n$) points to the first column index of row i in the array ja . That is, $ia(i)$ gives the index of the element in array a that contains the first non-zero block from row i of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zero blocks in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of BSR Format](#) for more details.

The array ia is accessed in all phases of the solution process.

Indexing of ia is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter $iparm(35)$. For zero-based indexing, the last element $ia(n+1)$ is assumed to be equal to the number of non-zero elements in matrix A .

NOTE

For centralized input ($iparm(40)=0$), provide the ia array at the master MPI process only. For distributed assembled input ($iparm(40)=1$ or $iparm(40)=2$), provide it at all MPI processes.

ja

INTEGER

For CSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal elements are stored (even if they are zeros) in the list of non-zero elements in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for $columns$ array in [Three Array Variation of CSR Format](#).

For BSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal blocks are stored (even if they are zeros) in the list of non-zero blocks in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for $columns$ array in [Three Array Variation of BSR Format](#).

The array ja is accessed in all phases of the solution process.

Indexing of ja is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter $iparm(35)$.

NOTE

For centralized input ($iparm(40)=0$), provide the ja array at the master MPI process only. For distributed assembled input ($iparm(40)=1$ or $iparm(40)=2$), provide it at all MPI processes.

$perm$

INTEGER

Ignored.

$nrhs$

INTEGER

Number of right-hand sides that need to be solved for.

$iparm$

INTEGER

Array, size 64. This array is used to pass various parameters to Parallel Direct Sparse Solver for Clusters Interface and to return some useful information after execution of the solver.

See [cluster_sparse_solver iparm Parameter](#) for more details about the $iparm$ parameters.

$msglvl$

INTEGER

Message level information. If $msglvl = 0$ then `cluster_sparse_solver` generates no output, if $msglvl = 1$ the solver prints statistical information to the screen.

Statistics include information such as the number of non-zero elements in *L-factor* and the timing for each phase.

Set *msglvl* = 1 if you report a problem with the solver, since the additional information provided can facilitate a solution.

<i>b</i>	<p>DOUBLE PRECISION - for real types of matrices (<i>mtype</i>=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=0)</p> <p>REAL - for real types of matrices (<i>mtype</i>=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=1)</p> <p>DOUBLE COMPLEX - for complex types of matrices (<i>mtype</i>=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=0)</p> <p>COMPLEX - for complex types of matrices (<i>mtype</i>=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=1)</p> <p>Array, size (<i>n</i>, <i>nrhs</i>). On entry, contains the right-hand side vector/matrix <i>B</i>, which is placed in memory contiguously. The <i>b</i>(<i>i</i>+(<i>k</i>-1)×<i>nrhs</i>) must hold the <i>i</i>-th component of <i>k</i>-th right-hand side vector. Note that <i>b</i> is only accessed in the solution phase.</p>
<i>comm</i>	<p>INTEGER</p> <p>MPI communicator. The solver uses the Fortran MPI communicator internally.</p>

Output Parameters

<i>pt</i>	Handle to internal data structure.
<i>perm</i>	Ignored.
<i>iparm</i>	<p>On output, some <i>iparm</i> values report information such as the numbers of non-zero elements in the factors.</p> <p>See cluster_sparse_solver iparm Parameter for more details about the <i>iparm</i> parameters.</p>
<i>b</i>	On output, the array is replaced with the solution if <i>iparm</i> (6) = 1.
<i>x</i>	<p>DOUBLE PRECISION - for real types of matrices (<i>mtype</i>=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=0)</p> <p>REAL - for real types of matrices (<i>mtype</i>=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=1)</p> <p>DOUBLE COMPLEX - for complex types of matrices (<i>mtype</i>=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=0)</p> <p>COMPLEX - for complex types of matrices (<i>mtype</i>=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters (<i>iparm</i>(28)=1)</p>

Array, size $(n, nrhs)$. If $iparm(6) = 0$ it contains solution vector/matrix X , which is placed contiguously in memory. The $x(i + (k-1) \times n)$ element must hold the i -th component of the k -th solution vector. Note that x is only accessed in the solution phase.

error

INTEGER

The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-5	unclassified (internal) error

cluster_sparse_solver iparm Parameter

The following table describes all individual components of the Parallel Direct Sparse Solver for Clusters Interface *iparm* parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description
<i>iparm</i> (1)	Use default values.
input	0 <i>iparm</i> (2) - <i>iparm</i> (64) are filled with default values. !=0 You must supply all values in components <i>iparm</i> (2) - <i>iparm</i> (64).
<i>iparm</i> (2)	Fill-in reducing ordering for the input matrix.
input	2* The nested dissection algorithm from the METIS package [Karypis98] . 3 The parallel version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Phase 1 takes significant time.
<i>iparm</i> (3)	Reserved. Set to zero.
<i>iparm</i> (6)	Write solution on x .
input	NOTE The array x is always used.
	0* The array x contains the solution; right-hand side vector b is kept unchanged.
	1 The solver stores the solution on the right-hand side b .
<i>iparm</i> (7)	Number of iterative refinement steps performed.
output	Reports the number of iterative refinement steps that were actually performed during the solve step.
<i>iparm</i> (8)	Iterative refinement step.
input	On entry to the solve and iterative refinement step, <i>iparm</i> (8) must be set to the maximum number of iterative refinement steps that the solver performs.

Component	Description
	<p>0* The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.</p>
	<p>>0 Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of <i>iparm(8)</i> steps of iterative refinement. The solver might stop the process before the maximum number of steps if</p> <ul style="list-style-type: none"> a satisfactory level of accuracy of the solution in terms of backward error is achieved, or if it determines that the required accuracy cannot be reached. In this case Parallel Direct Sparse Solver for Clusters Interface returns -4 in the <i>error</i> parameter. <p>The number of executed iterations is reported in <i>iparm(7)</i>.</p>
	<p><0 Same as above, but the accumulation of the residuum uses extended precision real and complex data types.</p> <p>Perturbed pivots result in iterative refinement (independent of <i>iparm(8)=0</i>) and the number of executed iterations is reported in <i>iparm(7)</i>.</p>
<i>iparm(9)</i>	Reserved. Set to zero.
<i>iparm(10)</i> input	<p>Pivoting perturbation.</p> <p>This parameter instructs Parallel Direct Sparse Solver for Clusters Interface how to handle small pivots or zero pivots for nonsymmetric matrices (<i>mtype</i> =11 or <i>mtype</i> =13) and symmetric matrices (<i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].</p> <p>Small pivots are perturbed with $\text{eps} = 10^{-iparm(10)}$.</p> <p>The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \text{eps} * A2 _{inf}$, where $\text{eps} = 10^{(-iparm(10))}$, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $A2 _{inf}$ is the infinity norm of the scaled and permuted matrix <i>A</i>. Any tiny pivots encountered during elimination are set to the sign (<i>l_II</i>)*$\text{eps} * A2 _{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{(-iparm(10))}$.</p>
	<p>13* The default value for nonsymmetric matrices(<i>mtype</i> =11, <i>mtype</i>=13), $\text{eps} = 10^{-13}$.</p>
	<p>8* The default value for symmetric indefinite matrices (<i>mtype</i> =-2, <i>mtype</i>=-4, <i>mtype</i>=6), $\text{eps} = 10^{-8}$.</p>
<i>iparm(11)</i> input	<p>Scaling vectors.</p> <p>Parallel Direct Sparse Solver for Clusters Interface uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale.</p> <p>Use <i>iparm(11)</i> = 1 (scaling) and <i>iparm(13)</i> = 1 (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (<i>phase</i>=11) you must provide the numerical values of the matrix <i>A</i> in array <i>a</i> in case of scaling and symmetric weighted matching.</p>
	<p>0* Disable scaling. Default for symmetric indefinite matrices.</p>
	<p>1* Enable scaling. Default for nonsymmetric matrices.</p>

Component	Description
	<p>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices ($mtype = 11, mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2, mtype = -4, mtype = 6$) when the symmetric weighted matchings are applied ($iparm(13) = 1$).</p> <p>Note that in the analysis phase ($phase=11$) you must provide the numerical values of the matrix A in case of scaling.</p>
$iparm(12)$	Reserved. Set to zero.
$iparm(13)$ input	<p>Improved accuracy using (non-) symmetric weighted matching.</p> <p>Parallel Direct Sparse Solver for Clusters Interface can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.</p>
	<p>0* Disable matching. Default for symmetric indefinite matrices.</p>
	<p>1* Enable matching. Default for nonsymmetric matrices.</p> <p>Maximum weighted matching algorithm to permute large elements close to the diagonal.</p> <p>It is recommended to use $iparm(11) = 1$ (scaling) and $iparm(13) = 1$ (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p> <p>Note that in the analysis phase ($phase=11$) you must provide the numerical values of the matrix A in case of symmetric weighted matching.</p>
$iparm(14)$ - $iparm(20)$	Reserved. Set to zero.
$iparm(21)$ input	<p>Pivoting for symmetric indefinite matrices.</p>
	<p>0 Apply 1x1 diagonal pivoting during the factorization process.</p>
	<p>1* Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of $mtype=-2, mtype=-4, or mtype=6$.</p>
$iparm(22)$ - $iparm(26)$	Reserved. Set to zero.
$iparm(27)$ input	<p>Matrix checker.</p>
	<p>0* Do not check the sparse matrix representation for errors.</p>
	<p>1 Check integer arrays ia and ja. In particular, check whether the column indices are sorted in increasing order within each row.</p>
$iparm(28)$ input	<p>Single or double precision Parallel Direct Sparse Solver for Clusters Interface.</p> <p>See $iparm(8)$ for information on controlling the precision of the refinement steps.</p>
	<p>0* Input arrays (a, x and b) and all internal arrays must be presented in double precision.</p>
	<p>1 Input arrays (a, x and b) must be presented in single precision.</p>

Component	Description
	In this case all internal computations are performed in single precision.
<i>iparm</i> (29) -	Reserved. Set to zero.
<i>iparm</i> (34)	
<i>iparm</i> (35)	One- or zero-based indexing of columns and rows.
input	<p>0* One-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 1 (Fortran-style indexing).</p> <p>1 Zero-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 0 (C-style indexing).</p>
<i>iparm</i> (37)	Format for matrix storage.
input	<p>0* Use CSR format (see Three Array Variation of BSR Format) for matrix storage.</p> <p>> 0 Use BSR format (see Three Array Variation of BSR Format) for matrix storage with blocks of size <i>iparm</i>(37).</p>
<p>NOTE Intel MKL does not support BSR format in these cases:</p> <ul style="list-style-type: none"> <i>iparm</i>(11) > 0 Scaling vectors <i>iparm</i>(13) > 0 Weighted matching <i>iparm</i>(31) > 0 Partial solution <i>iparm</i>(36) > 0 Schur complement <i>iparm</i>(56) > 0 Pivoting control <i>iparm</i>(60) > 0 OOC Intel MKL PARDISO 	
<i>iparm</i> (38) -	Reserved. Set to zero.
<i>iparm</i> (39)	
<i>iparm</i> (40)	Matrix input format.
input	<p>NOTE Performance of the reordering step of the Parallel Direct Sparse Solver for Clusters Interface is slightly better for assembled format (CSR, <i>iparm</i>(40) = 0) than for distributed format (DCSR, <i>iparm</i>(40) > 0) for the same matrices, so if the matrix is assembled on one node do not distribute it before calling <code>cluster_sparse_solver</code>.</p> <p>0* Provide the matrix in usual centralized input format: the master MPI process stores all data from matrix <i>A</i>, with rank=0.</p> <p>1 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix <i>A</i> data. Set the bounds of the domain using iparm(41) and iparm(42). The solution vector is placed on the master process.</p>

Component	Description
	2 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <i>iparm(41)</i> and <i>iparm(42)</i> . The solution vector, A , and RHS elements are distributed between processes in same manner.
	3 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <i>iparm(41)</i> and <i>iparm(42)</i> . The A and RHS elements are distributed between processes in same manner and the solution vector is the same on each process
<i>iparm(41)</i> input	Beginning of input domain. The number of the matrix A row, RHS element, and, for <i>iparm(40)</i> =2, solution vector that begins the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format (<i>iparm(40)</i> > 0). See Sparse Matrix Storage Formats for more details.
<i>iparm(42)</i> input	End of input domain. The number of the matrix A row, RHS element, and, for <i>iparm(40)</i> =2, solution vector that ends the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format (<i>iparm(40)</i> > 0). See Sparse Matrix Storage Formats for more details.
<i>iparm(43)</i> -	Reserved. Set to zero.
<i>iparm(64)</i> input	

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Direct Sparse Solver (DSS) Interface Routines

Intel MKL supports the DSS interface, an alternative to the Intel MKL PARDISO interface for the direct sparse solver. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and utilizes the general scheme described in [Appendix A Linear Solvers Basics](#) for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The DSS interface also supports the out-of-core (OOC) mode.

Table "DSS Interface Routines" lists the names of the routines and describes their general use.

DSS Interface Routines

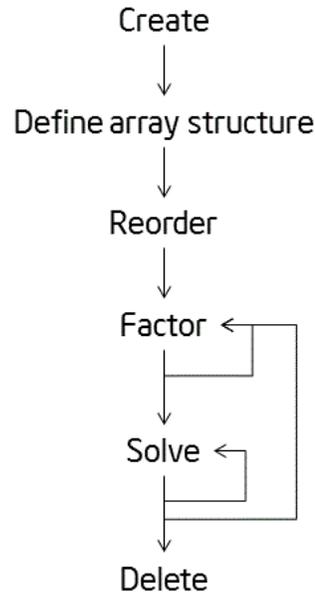
Routine	Description
<code>dss_create</code>	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
<code>dss_define_structure</code>	Informs the solver of the locations of the non-zero elements of the matrix.
<code>dss_reorder</code>	Based on the non-zero structure of the matrix, computes a permutation vector to reduce fill-in during the factoring process.
<code>dss_factor_real</code> , <code>dss_factor_complex</code>	Computes the LU , LDL^T or LL^T factorization of a real or complex matrix.
<code>dss_solve_real</code> , <code>dss_solve_complex</code>	Computes the solution vector for a system of equations based on the factorization computed in the previous phase.
<code>dss_delete</code>	Deletes all data structures created during the solving process.
<code>dss_statistics</code>	Returns statistics about various phases of the solving process.
<code>mkl_cvt_to_null_terminated_str</code>	Passes character strings from Fortran routines to C routines.

To find a single solution vector for a single system of equations with a single right-hand side, invoke the Intel MKL DSS interface routines in this order:

1. `dss_create`
2. `dss_define_structure`
3. `dss_reorder`
4. `dss_factor_real`, `dss_factor_complex`
5. `dss_solve_real`, `dss_solve_complex`
6. `dss_delete`

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is sometimes necessary to invoke the Intel MKL sparse routines in an order other than that listed, which is possible using the DSS interface. The solving process is conceptually divided into six phases. [Figure "Typical order for invoking DSS interface routines"](#) indicates the typical order in which the DSS interface routines can be invoked.

Typical order for invoking DSS interface routines



See the code examples that use the DSS interface routines to solve systems of linear equations in the Intel MKL installation directory (`dss_*.f`).

- `examples/solverf/source`

DSS Interface Description

Each DSS routine reads from or writes to a data object called a *handle*. Refer to [Memory Allocation and Handles](#) to determine the correct method for declaring a handle argument for each language. For simplicity, the descriptions in DSS routines refer to the data type as `MKL_DSS_HANDLE`.

Routine Options

The DSS routines have an integer argument (referred below to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see [Implementation Details](#)). The routines accept options for setting the message and termination levels as described in [Table "Symbolic Names for the Message and Termination Levels Options"](#). Additionally, each routine accepts the option `MKL_DSS_DEFAULTS` that sets the default values (as documented) for *opt* to the routine.

Symbolic Names for the Message and Termination Levels Options

Message Level	Termination Level
<code>MKL_DSS_MSG_LVL_SUCCESS</code>	<code>MKL_DSS_TERM_LVL_SUCCESS</code>
<code>MKL_DSS_MSG_LVL_INFO</code>	<code>MKL_DSS_TERM_LVL_INFO</code>
<code>MKL_DSS_MSG_LVL_WARNING</code>	<code>MKL_DSS_TERM_LVL_WARNING</code>
<code>MKL_DSS_MSG_LVL_ERROR</code>	<code>MKL_DSS_TERM_LVL_ERROR</code>
<code>MKL_DSS_MSG_LVL_FATAL</code>	<code>MKL_DSS_TERM_LVL_FATAL</code>

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

You can specify both message and termination level for a DSS routine by adding the options together. For example, to set the message level to `debug` and the termination level to `error` for all the DSS routines, use the following call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix.

CAUTION

Do not modify the contents of these arrays after they are passed to one of the solver routines.

DSS Implementation Details

To promote portability across platforms and ease of use across different languages, use the appropriate Intel MKL DSS header file:

- `mk1_dss.f77` for F77 programs
- `mk1_dss.f90` for F90 programs

The header file defines symbolic constants for returned error values, function options, certain defined data types, and function prototypes.

NOTE

Constants for options, returned error values, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel MKL DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

You do not need to allocate any temporary working storage in order to use the Intel MKL DSS routines, because the solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a *handle* data object.

Each of the Intel MKL DSS routines creates, uses, or deletes a handle. Consequently, any program calling an Intel MKL DSS routine must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that must be used when declaring a handle object in your code.

- Fortran 90:

```
INCLUDE "mk1_dss.f90"  
TYPE(MKL_DSS_HANDLE) handle
```

- FORTRAN 77 (compilers that support eight byte integers):

```
INCLUDE "mk1_dss.f77"  
INTEGER*8 handle
```

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the returned error values
- user options for the solver routines

- constants indicating message severity.

DSS Routines

dss_create

Initializes the solver.

Syntax

```
call dss_create(handle, opt)
```

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The `dss_create` routine initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel MKL DSS routines must use the value of the `handle` returned by `dss_create`.

WARNING

Do not write the value of `handle` directly.

The default value of the parameter `opt` is

```
MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.
```

By default, the DSS routines use double precision for solving systems of linear equations. The precision used by the DSS routines can be set to single mode by adding the following value to the `opt` parameter:

```
MKL_DSS_SINGLE_PRECISION.
```

Input data and internal arrays are required to have single precision.

By default, the DSS routines use Fortran style (one-based) indexing for input arrays of integer types (the first value is referenced as array element 1). To set indexing to C style (the first value is referenced as array element 0), add the following value to the `opt` parameter:

```
MKL_DSS_ZERO_BASED_INDEXING.
```

The `opt` parameter can also control number of refinement steps used on the solution stage by specifying the two following values:

```
MKL_DSS_REFINEMENT_OFF - maximum number of refinement steps is set to zero;
```

```
MKL_DSS_REFINEMENT_ON (default value) - maximum number of refinement steps is set to 2.
```

By default, DSS uses in-core computations. To launch the out-of-core version of DSS (OOC DSS) you can add to this parameter one of two possible values: `MKL_DSS_OOC_STRONG` and `MKL_DSS_OOC_VARIABLE`.

```
MKL_DSS_OOC_STRONG - OOC DSS is used.
```

```
MKL_DSS_OOC_VARIABLE - if the memory needed for the matrix factors is less than the value of the environment variable MKL_PARDISO_OOC_MAX_CORE_SIZE, then the OOC DSS uses the in-core kernels of Intel MKL PARDISO, otherwise it uses the OOC computations.
```

The variable `MKL_PARDISO_OOC_MAX_CORE_SIZE` defines the maximum size of RAM allowed for storing work arrays associated with the matrix factors. It is ignored if `MKL_DSS_OOC_STRONG` is set. The default value of `MKL_PARDISO_OOC_MAX_CORE_SIZE` is 2000 MB. This value and default path and file name for storing temporary data can be changed using the configuration file `pardiso_ooc.cfg` or command line (See more details in the description of the [pardiso](#) routine).

WARNING

Other than message and termination level options, do not change the OOC DSS settings after they are specified in the routine `dss_create`.

Input Parameters

opt INTEGER, INTENT(IN)
Parameter to pass the DSS options. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle TYPE (MKL_DSS_HANDLE), INTENT(OUT)
Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS
MKL_DSS_INVALID_OPTION
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

dss_define_structure

Communicates locations of non-zero elements in the matrix to the solver.

Syntax

```
call dss_define_structure(handle, opt, rowIndex, nRows, nCols, columns, nNonZeros);
```

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The routine `dss_define_structure` communicates the locations of the *nNonZeros* number of non-zero elements in a matrix of *nRows* * *nCols* size to the solver.

NOTE

The Intel MKL DSS software operates only on square matrices, so *nRows* must be equal to *nCols*.

To communicate the locations of non-zero elements in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying the value for the options argument *opt*. You can set the following values for real matrices:
 - `MKL_DSS_SYMMETRIC_STRUCTURE`
 - `MKL_DSS_SYMMETRIC`

- MKL_DSS_NON_SYMMETRIC

and for complex matrices:

- MKL_DSS_SYMMETRIC_STRUCTURE_COMPLEX
- MKL_DSS_SYMMETRIC_COMPLEX
- MKL_DSS_NON_SYMMETRIC_COMPLEX

The information about the matrix type must be defined in `dss_define_structure`.

2. Provide the actual locations of the non-zeros by means of the arrays `rowIndex` and `columns` (see [Sparse Matrix Storage Format](#)).

Input Parameters

<code>opt</code>	INTEGER, INTENT (IN)	Parameter to pass the DSS options. The default value for the matrix structure is MKL_DSS_SYMMETRIC.
<code>rowIndex</code>	INTEGER, INTENT (IN)	Array of size <code>nRows+1</code> . Defines the location of non-zero entries in the matrix.
<code>nRows</code>	INTEGER, INTENT (IN)	Number of rows in the matrix.
<code>nCols</code>	INTEGER, INTENT (IN)	Number of columns in the matrix; must be equal to <code>nRows</code> .
<code>columns</code>	INTEGER, INTENT (IN)	Array of size <code>nNonZeros</code> . Defines the column location of non-zero entries in the matrix.
<code>nNonZeros</code>	INTEGER, INTENT (IN)	Number of non-zero elements in the matrix.

Output Parameters

<code>handle</code>	TYPE (MKL_DSS_HANDLE), INTENT (INOUT)	Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
---------------------	---------------------------------------	--

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_STRUCTURE_ERR
 MKL_DSS_ROW_ERR
 MKL_DSS_COL_ERR
 MKL_DSS_NOT_SQUARE
 MKL_DSS_TOO_FEW_VALUES

MKL_DSS_TOO_MANY_VALUES

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_reorder

Computes or sets a permutation vector that minimizes the fill-in during the factorization phase.

Syntax

```
call dss_reorder(handle, opt, perm)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

If *opt* contains the option MKL_DSS_AUTO_ORDER, then the routine `dss_reorder` computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_METIS_OPENMP_ORDER, then the routine `dss_reorder` computes permutation vector using the parallel nested dissections algorithm to minimize the fill-in during the factorization phase. This option can be used to decrease the time of `dss_reorder` call on multi-core computers. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_MY_ORDER, then you must supply a permutation vector in the array *perm*. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to `dss_define_structure`.

If *opt* contains the option MKL_DSS_GET_ORDER, then the permutation vector computed during the `dss_reorder` call is copied to the array *perm*. In this case you must allocate the array *perm* beforehand. The permutation vector is computed in the same way as if the option MKL_DSS_AUTO_ORDER is set.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

<i>opt</i>	INTEGER, INTENT (IN)
	Parameter to pass the DSS options. The default value for the permutation type is MKL_DSS_AUTO_ORDER.
<i>perm</i>	INTEGER, INTENT (IN)

Array of length $nRows$. Contains a user-defined permutation vector (accessed only if *opt* contains MKL_DSS_MY_ORDER or MKL_DSS_GET_ORDER).

Output Parameters

handle TYPE (MKL_DSS_HANDLE), INTENT (INOUT)
 Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_REORDER_ERR
 MKL_DSS_REORDER1_ERR
 MKL_DSS_I32BIT_ERR
 MKL_DSS_FAILURE
 MKL_DSS_OUT_OF_MEMORY
 MKL_DSS_MSG_LVL_ERR
 MKL_DSS_TERM_LVL_ERR

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location of non-zero elements.

Syntax

```
call dss_factor_real(handle, opt, rValues)
call dss_factor_complex(handle, opt, cValues)
call dss_factor(handle, opt, Values)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to [dss_define_structure](#) and whose non-zero values are given in the array *rValues*, *cValues* or *Values*. Data type These arrays must be of length *nNonZeros* as defined in a previous call to [dss_define_structure](#).

NOTE

The data type (single or double precision) of *rValues*, *cValues*, *Values* must be in correspondence with precision specified by the parameter *opt* in the routine [dss_create](#).

The *opt* argument can contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE
- MKL_DSS_INDEFINITE
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE
- MKL_DSS_HERMITIAN_INDEFINITE

depending on your matrix's type.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>handle</i>	TYPE (MKL_DSS_HANDLE), INTENT (INOUT)	Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
<i>opt</i>	INTEGER, INTENT (IN)	Parameter to pass the DSS options. The default value is MKL_DSS_POSITIVE_DEFINITE.
<i>rValues</i>	REAL*8 REAL (KIND=4), INTENT (IN) or REAL (KIND=8), INTENT (IN)	Array of elements of the matrix <i>A</i> . Real data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i> .
<i>cValues</i>	COMPLEX*16 COMPLEX (KIND=4), INTENT (IN) or COMPLEX (KIND=8), INTENT (IN)	Array of elements of the matrix <i>A</i> . Complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i> .
<i>Values</i>	REAL (KIND=4), INTENT (OUT), or REAL (KIND=8), INTENT (OUT), or COMPLEX (KIND=4), INTENT (OUT), or COMPLEX (KIND=8), INTENT (OUT)	Array of elements of the matrix <i>A</i> . Real or complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i> .

Return Values

- MKL_DSS_SUCCESS
- MKL_DSS_STATE_ERR
- MKL_DSS_INVALID_OPTION
- MKL_DSS_OPTION_CONFLICT
- MKL_DSS_VALUES_ERR
- MKL_DSS_OUT_OF_MEMORY

MKL_DSS_ZERO_PIVOT
 MKL_DSS_FAILURE
 MKL_DSS_MSG_LVL_ERR
 MKL_DSS_TERM_LVL_ERR
 MKL_DSS_OOC_MEM_ERR
 MKL_DSS_OOC_OC_ERR
 MKL_DSS_OOC_RW_ERR

dss_solve_real, dss_solve_complex

Compute the corresponding solution vector and place it in the output array.

Syntax

```
call dss_solve_real(handle, opt, rRhsValues, nRhs, rSolValues)
call dss_solve_complex(handle, opt, cRhsValues, nRhs, cSolValues)
call dss_solve(handle, opt, RhsValues, nRhs, SolValues)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

For each right-hand side column vector defined in the arrays *rRhsValues*, *cRhsValues*, or *RhsValues*, these routines compute the corresponding solution vector and place it in the arrays *rSolValues*, *cSolValues*, or *SolValues* respectively.

NOTE

The data type (single or double precision) of all arrays must be in correspondence with precision specified by the parameter *opt* in the routine *dss_create*.

The lengths of the right-hand side and solution vectors, *nCols* and *nRows* respectively, must be defined in a previous call to [dss_define_structure](#).

By default, both routines perform the full solution step (it corresponds to *phase* = 33 in Intel MKL PARDISO). The parameter *opt* enables you to calculate the final solution step-by-step, calling forward and backward substitutions.

If it is set to `MKL_DSS_FORWARD_SOLVE`, the forward substitution (corresponding to *phase* = 331 in Intel MKL PARDISO) is performed;

if it is set to `MKL_DSS_DIAGONAL_SOLVE`, the diagonal substitution (corresponding to *phase* = 332 in Intel MKL PARDISO) is performed, if possible;

if it is set to `MKL_DSS_BACKWARD_SOLVE`, the backward substitution (corresponding to *phase* = 333 in Intel MKL PARDISO) is performed.

For more details about using these substitutions for different types of matrices, see [Separate Forward and Backward Substitution](#) in the Intel MKL PARDISO solver description.

This parameter also can control the number of refinement steps that is used on the solution stage: if it is set to `MKL_DSS_REFINEMENT_OFF`, the maximum number of refinement steps equal to zero, and if it is set to `MKL_DSS_REFINEMENT_ON` (default value), the maximum number of refinement steps is equal to 2.

MKL_DSS_CONJUGATE_SOLVE option added to the parameter *opt* enables solving a conjugate transposed system $A^T X = B$ based on the factorization of the matrix *A*. This option is equivalent to the parameter *iparm*(12) = 1 in Intel MKL PARDISO.

MKL_DSS_TRANSPOSE_SOLVE option added to the parameter *opt* enables solving a transposed system $A^T X = B$ based on the factorization of the matrix *A*. This option is equivalent to the parameter *iparm*(12) = 2 in Intel MKL PARDISO.

Input Parameters

<i>handle</i>	<p>TYPE (MKL_DSS_HANDLE), INTENT (INOUT)</p> <p>Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).</p>
<i>opt</i>	<p>INTEGER, INTENT (IN)</p> <p>Parameter to pass the DSS options.</p>
<i>nRhs</i>	<p>INTEGER, INTENT (IN)</p> <p>Number of the right-hand sides in the system of linear equations.</p>
<i>rRhsValues</i>	<p>REAL*8</p> <p>REAL (KIND=4), INTENT (IN) or</p> <p>REAL (KIND=8), INTENT (IN)</p> <p>Array of size <i>nRows</i> * <i>nRhs</i>. Contains real right-hand side vectors. Real data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i>.</p>
<i>cRhsValues</i>	<p>COMPLEX*16</p> <p>COMPLEX (KIND=4), INTENT (IN) or</p> <p>COMPLEX (KIND=8), INTENT (IN)</p> <p>Array of size <i>nRows</i> * <i>nRhs</i>. Contains complex right-hand side vectors. Complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i>.</p>
<i>RhsValues</i>	<p>REAL (KIND=4), INTENT (IN), or</p> <p>REAL (KIND=8), INTENT (IN), or</p> <p>COMPLEX (KIND=4), INTENT (IN), or</p> <p>COMPLEX (KIND=8), INTENT (IN)</p> <p>Array of size <i>nRows</i> * <i>nRhs</i>. Contains right-hand side vectors. Real or complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i>.</p>

Output Parameters

<i>rSolValues</i>	<p>REAL (KIND=4), INTENT (OUT) or</p> <p>REAL (KIND=8), INTENT (OUT)</p>
-------------------	--

Array of size $nCols * nRhs$. Contains real solution vectors. Real data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

cSolValues

COMPLEX (KIND=4), INTENT (OUT) or

COMPLEX (KIND=8), INTENT (OUT)

Array of size $nCols * nRhs$. Contains complex solution vectors. Complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

SolValues

REAL (KIND=4), INTENT (OUT), or

REAL (KIND=8), INTENT (OUT), or

COMPLEX (KIND=4), INTENT (OUT), or

COMPLEX (KIND=8), INTENT (OUT)

Array of size $nCols * nRhs$. Contains solution vectors. Real or complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_DIAG_ERR

MKL_DSS_FAILURE

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

MKL_DSS_OOC_MEM_ERR

MKL_DSS_OOC_OC_ERR

MKL_DSS_OOC_RW_ERR

dss_delete

Deletes all of the data structures created during the solutions process.

Syntax

call `dss_delete(handle, opt)`

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The routine `dss_delete` deletes all data structures created during the solving process.

Input Parameters

opt INTEGER, INTENT (IN)
Parameter to pass the DSS options. The default value is MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.

Output Parameters

handle TYPE (MKL_DSS_HANDLE), INTENT (INOUT)
Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS
MKL_DSS_STATE_ERR
MKL_DSS_INVALID_OPTION
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

```
call dss_statistics(handle, opt, statArr, retValues)
```

Include Files

- mkl.fi, mkl_dss.f90

Description

The `dss_statistics` routine returns statistics about various phases of the solving process. This routine gathers the following statistics:

- time taken to do reordering,
- time taken to do factorization,
- duration of problem solving,
- determinant of the symmetric indefinite input matrix,
- inertia of the symmetric indefinite input matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter `statArr`. The value of the statistics is returned in double precision in a return array, which you must allocate beforehand.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics can only be requested at the appropriate stages of the solving process. For example, requesting `FactorTime` before a matrix is factored leads to an error.

The following table shows the point at which each individual statistics item can be requested:

Statistics Calling Sequences

Type of Statistics	When to Call
<code>ReorderTime</code>	After <code>dss_reorder</code> is completed successfully.
<code>FactorTime</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>SolveTime</code>	After <code>dss_solve_real</code> or <code>dss_solve_complex</code> is completed successfully.
<code>Determinant</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>Inertia</code>	After <code>dss_factor_real</code> is completed successfully and the matrix is real, symmetric, and indefinite.
<code>Flops</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
<code>Peakmem</code>	After <code>dss_reorder</code> is completed successfully.
<code>Factormem</code>	After <code>dss_reorder</code> is completed successfully.
<code>Solvemem</code>	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.

Input Parameters

<code>handle</code>	<p>TYPE (MKL_DSS_HANDLE), INTENT (IN)</p> <p>Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).</p>								
<code>opt</code>	<p>INTEGER, INTENT (IN)</p> <p>Parameter to pass the DSS options.</p>								
<code>statArr</code>	<p>INTEGER, INTENT (IN)</p> <p>Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect):</p> <table style="margin-left: 20px;"> <tbody> <tr> <td><code>ReorderTime</code></td> <td>Amount of time taken to do the reordering.</td> </tr> <tr> <td><code>FactorTime</code></td> <td>Amount of time taken to do the factorization.</td> </tr> <tr> <td><code>SolveTime</code></td> <td>Amount of time taken to solve the problem after factorization.</td> </tr> <tr> <td><code>Determinant</code></td> <td> <p>Determinant of the matrix A.</p> <p>For real matrices: the determinant is returned as <code>det_pow</code>, <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$.</p> <p>For complex matrices: the determinant is returned as <code>det_pow</code>, <code>det_re</code>, <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.</p> </td> </tr> </tbody> </table>	<code>ReorderTime</code>	Amount of time taken to do the reordering.	<code>FactorTime</code>	Amount of time taken to do the factorization.	<code>SolveTime</code>	Amount of time taken to solve the problem after factorization.	<code>Determinant</code>	<p>Determinant of the matrix A.</p> <p>For real matrices: the determinant is returned as <code>det_pow</code>, <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$.</p> <p>For complex matrices: the determinant is returned as <code>det_pow</code>, <code>det_re</code>, <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.</p>
<code>ReorderTime</code>	Amount of time taken to do the reordering.								
<code>FactorTime</code>	Amount of time taken to do the factorization.								
<code>SolveTime</code>	Amount of time taken to solve the problem after factorization.								
<code>Determinant</code>	<p>Determinant of the matrix A.</p> <p>For real matrices: the determinant is returned as <code>det_pow</code>, <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$.</p> <p>For complex matrices: the determinant is returned as <code>det_pow</code>, <code>det_re</code>, <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.</p>								

Inertia	<p>Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z), where p is the number of positive eigenvalues, n is the number of negative eigenvalues, and z is the number of zero eigenvalues.</p> <p>Inertia is returned as three consecutive return array locations p, n, z.</p> <p>Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0.</p> <p>Inertia of a k-by-k real symmetric positive definite matrix is always $(k, 0, 0)$. Therefore Inertia is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.</p>
Flops	Number of floating point operations performed during the factorization.
Peakmem	Total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.
Factormem	Permanent memory in kilobytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.
Solvemem	Total double precision memory consumption (kilobytes) of the solver for the factorization and solve phases.

NOTE

To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of [mkl_cvt_to_null_terminated_str](#) for details.

Output Parameters

`retValues` REAL (KIND=8), INTENT (OUT)
Value of the statistics returned.

Finding 'time used to reorder' and 'inertia' of a matrix

The example below illustrates the use of the `dss_statistics` routine.

To find the above values, call `dss_statistics(handle, opt, statArr, retValue)`, where `statArr` is "ReorderTime,Inertia"

In this example, `retValue` has the following values:

`retValue[0]` Time to reorder.

<code>retValue[1]</code>	Positive Eigenvalues.
<code>retValue[2]</code>	Negative Eigenvalues.
<code>retValue[3]</code>	Zero Eigenvalues.

Return Values

`MKL_DSS_SUCCESS`
`MKL_DSS_INVALID_OPTION`
`MKL_DSS_STATISTICS_INVALID_MATRIX`
`MKL_DSS_STATISTICS_INVALID_STATE`
`MKL_DSS_STATISTICS_INVALID_STRING`
`MKL_DSS_MSG_LVL_ERR`
`MKL_DSS_TERM_LVL_ERR`

`mkl_cvt_to_null_terminated_str`

Passes character strings from Fortran routines to C routines.

Syntax

`mkl_cvt_to_null_terminated_str (destStr, destLen, srcStr)`

Include Files

- `mkl.fi`, `mkl_dss.f90`

Description

The routine `mkl_cvt_to_null_terminated_str` passes character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Input Parameters

<code>destLen</code>	INTEGER. Length of the output array <code>destStr</code> .
<code>srcStr</code>	STRING. Input string.

Output Parameters

<code>destStr</code>	INTEGER. One-dimensional array of integers.
----------------------	---

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

Intel MKL supports iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as the RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI conjugate gradient solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI flexible generalized minimal residual solver, or RCI FGMRES) of linear algebraic equations. This interface uses the general RCI scheme described in [Dong95].

See the [Appendix A Linear Solvers Basics](#) for discussion of terms and concepts related to the ISS routines.

The term *RCI* indicates that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user performs them and passes the result to the solver. This makes the solver more universal as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines described in [Sparse BLAS Level 2 and Level 3 Routines](#).

NOTE

The RCI CG solver is implemented in two versions: for system of equations with a single right-hand side, and for systems of equations with multiple right-hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and not positive definite.

The FGMRES method may fail if the matrix is degenerate.

Table "RCI CG Interface Routines" lists the names of the routines, and describes their general use.

RCI ISS Interface Routines

Routine	Description
dcg_init , dcmrhs_init , dfgmres_init	Initializes the solver.
dcg_check , dcmrhs_check , dfgmres_check	Checks the consistency and correctness of the user defined data.
dcg , dcmrhs , dfgmres	Computes the approximate solution vector.
dcg_get , dcmrhs_get , dfgmres_get	Retrieves the number of the current iteration.

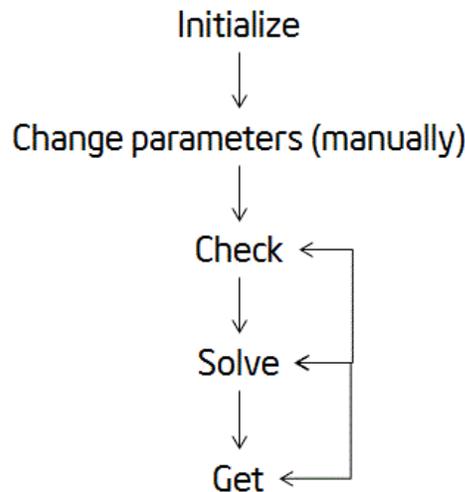
The Intel MKL RCI ISS interface routines are normally invoked in this order:

1. `<system_type>_init`
2. `<system_type>_check`
3. `<system_type>`
4. `<system_type>_get`

Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram indicates the typical order in which the RCI ISS interface routines are invoked.

Typical Order for Invoking RCI ISS interface Routines



See the code examples that use the RCI ISS interface routines to solve systems of linear equations in the Intel MKL installation directory.

- `examples/solverf/source`

CG Interface Description

All types in this documentation refer to the common Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right-hand side (SRHS), and for a system of equations with multiple right-hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.

Routine Options

All of the RCI CG routines have common parameters for passing various options to the routines (see [CG Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `dcg` to compute the solution of a system of linear algebraic equations. The Intel MKL RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `dcg_init/dcgmrs_init` routine.

<code>n</code>	<code>INTEGER</code> , this parameter sets the size of the problem in the <code>dcg_init/dcgmrs_init</code> routine. All the other routines use the <code>ipar(1)</code> parameter instead. Note that the coefficient matrix A is a square matrix of size $n*n$.
<code>x</code>	<code>DOUBLE PRECISION</code> array of size n for SRHS, or matrix of size $(n*nrhs)$ for MRHS. This parameter contains the current approximation to the solution. Before the first call to the <code>dcg/dcgmrs</code> routine, it contains the initial approximation to the solution.
<code>nrhs</code>	<code>INTEGER</code> , this parameter sets the number of right-hand sides for MRHS routines.
<code>b</code>	<code>DOUBLE PRECISION</code> array containing a single right-hand side vector, or matrix of size $n*nrhs$ containing right-hand side vectors.
<code>RCI_request</code>	<code>INTEGER</code> , this parameter gives information about the result of work of the RCI CG routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:

<code>RCI_request= 1</code>	multiply the matrix by <code>tmp(1:n,1)</code> , put the result in <code>tmp(1:n,2)</code> , and return the control to the <code>dcg/dcgmrhs</code> routine;
<code>RCI_request= 2</code>	to perform the stopping tests. If they fail, return the control to the <code>dcg/dcgmrhs</code> routine. If the stopping tests succeed, it indicates that the solution is found and stored in the <code>x</code> array;
<code>RCI_request= 3</code>	for SRHS: apply the preconditioner to <code>tmp(1:n,3)</code> , put the result in <code>tmp(1:n,4)</code> , and return the control to the <code>dcg</code> routine; for MRHS: apply the preconditioner to <code>tmp(:, 3+ipar(3))</code> , put the result in <code>tmp(:,3)</code> , and return the control to the <code>dcgmrhs</code> routine.

Note that the `dcg_get/dcgmrhs_get` routine does not change the parameter `RCI_request`. This enables use of this routine inside the *reverse communication* computations.

ipar

INTEGER array, of size 128 for SRHS, and of size $(128+2*nrhs)$ for MRHS. This parameter specifies the integer set of data for the RCI CG computations:

<code>ipar(1)</code>	specifies the size of the problem. The <code>dcg_init/dcgmrhs_init</code> routine assigns <code>ipar(1)=n</code> . All the other routines use this parameter instead of <code>n</code> . There is no default value for this parameter.
<code>ipar(2)</code>	specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise, the error and warning messages are written to the newly created files <code>dcg_errors.txt</code> and <code>dcg_check_warnings.txt</code> , respectively. Note that if <code>ipar(6)</code> and <code>ipar(7)</code> parameters are set to 0, error and warning messages are not generated at all.
<code>ipar(3)</code>	for SRHS: contains the current stage of the RCI CG computations. The initial value is 1; for MRHS: contains the number of the right-hand side for which the calculations are currently performed.

WARNING

Avoid altering this variable during computations.

<code>ipar(4)</code>	contains the current iteration number. The initial value is 0.
----------------------	--

<i>ipar</i> (5)	specifies the maximum number of iterations. The default value is $\min(150, n)$.
<i>ipar</i> (6)	if the value is not equal to 0, the routines output error messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output error messages at all, but return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (7)	if the value is not equal to 0, the routines output warning messages in accordance with the parameter <i>ipar</i> (2). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar</i> (8)	if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine performs the stopping test for the maximum number of iterations: $ipar(4) \leq ipar(5)$. Otherwise, the method is stopped and the corresponding value is assigned to the <i>RCI_request</i> . If the value is 0, the routine does not perform this stopping test. The default value is 1.
<i>ipar</i> (9)	if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine performs the residual stopping test: $dpar(5) \leq dpar(4) = dpar(1) * dpar(3) + dpar(2)$. Otherwise, the method is stopped and corresponding value is assigned to the <i>RCI_request</i> . If the value is 0, the routine does not perform this stopping test. The default value is 0.
<i>ipar</i> (10)	if the value is not equal to 0, the <i>dcg/dcgmrhs</i> routine requests a user-defined stopping test by setting the output parameter <i>RCI_request</i> =2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.
<p>NOTE At least one of the parameters <i>ipar</i>(8) - <i>ipar</i>(10) must be set to 1.</p>	
<i>ipar</i> (11)	if the value is equal to 0, the <i>dcg/dcgmrhs</i> routine runs the non-preconditioned version of the corresponding CG method. Otherwise, the routine runs the preconditioned version of the CG method, and by setting the output parameter <i>RCI_request</i> =3, indicates that you must perform the preconditioning step. The default value is 0.
<i>ipar</i> (12:128)	are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128 for a single right-hand side.

ipar(12:128+2**nrhs*) are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128+2**nrhs* for multiple right-hand sides.

dpar

DOUBLE PRECISION array, for SRHS of size 128, for MRHS of size (128+2**nrhs*); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

- dpar*(1) specifies the relative tolerance. The default value is 1.0×10^{-6} .
- dpar*(2) specifies the absolute tolerance. The default value is 0.0.
- dpar*(3) specifies the square norm of the initial residual (if it is computed in the *dcg/dcgmrhs* routine). The initial value is 0.0.
- dpar*(4) service variable equal to $dpar(1) * dpar(3) + dpar(2)$ (if it is computed in the *dcg/dcgmrhs* routine). The initial value is 0.0.
- dpar*(5) specifies the square norm of the current residual. The initial value is 0.0.
- dpar*(6) specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
- dpar*(7) contains the *alpha* parameter of the CG method. The initial value is 0.0.
- dpar*(8) contains the *beta* parameter of the CG method, it is equal to $dpar(5) / dpar(6)$ The initial value is 0.0.
- dpar*(9:128) are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *dpar* with length 128 for a single right-hand side.

dpar(9:128+2**nrhs*) are reserved for internal use in the current RCI CG
(9:128 + 2**nrhs*) SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *dpar* with length 128+2**nrhs* for multiple right-hand sides.

tmp

DOUBLE PRECISION array of size (*n*, 4) for SRHS, and (*n*, (3+*nrhs*)) for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

<i>tmp</i> (:, 1)	specifies the current search direction. The initial value is 0.0.
<i>tmp</i> (:, 2)	contains the matrix multiplied by the current search direction. The initial value is 0.0.
<i>tmp</i> (:, 3)	contains the current residual. The initial value is 0.0.
<i>tmp</i> (:, 4)	contains the inverse of the preconditioner applied to the current residual for the SRHS version of CG. There is no initial value for this parameter.
<i>tmp</i> (:, 4:3+ <i>nrhs</i>)	contains the inverse of the preconditioner applied to the current residual for the MRHS version of CG. There is no initial value for this parameter.

NOTE

You can define this array in the code using RCI CG SRHS as `DOUBLE PRECISION tmp(n, 3)` if you run only non-preconditioned CG iterations.

Schemes of Using the RCI CG Routines

The following pseudocode shows the general schemes of using the RCI CG routines for the SRHS case. The MRHS is similar (see the example code for more details).

```

...
generate matrix A
generate preconditioner C (optional)
  call dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
  change parameters in ipar, dpar if necessary
  call dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)

```

```

1 call dcg(n, x, b, RCI_request, ipar, dpar, tmp)
  if (RCI_request.eq.1) then
    multiply the matrix A by tmp(1:n,1) and put the result in tmp(1:n,2)
    It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation
c  proceed with CG iterations
    goto 1
  endif
  if (RCI_request.eq.2) then
    do the stopping test
    if (test not passed) then
c  proceed with CG iterations
    go to 1
    else
c  stop CG iterations
    goto 2
  endif
  endif
  if (RCI_request.eq.3) then (optional)
    apply the preconditioner C inverse to tmp(1:n,3) and put the result in tmp(1:n,4)
c  proceed with CG iterations
    goto 1
  end
2 call dcg_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
  current iteration number is in itercount
  the computed approximation is in the array x

```

FGMRES Interface Description

All types in this documentation refer to the common Fortran types: INTEGER and DOUBLE PRECISION.

Routine Options

All of the RCI FGMRES routines have common parameters for passing various options to the routines (see [FGMRES Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine `dfgmres` to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `dfgmres_init` routine.

<i>n</i>	INTEGER, this parameter sets the size of the problem in the <code>dfgmres_init</code> routine. All the other routines use the <code>ipar(1)</code> parameter instead. Note that the coefficient matrix <i>A</i> is a square matrix of size $n*n$.								
<i>x</i>	DOUBLE PRECISION array, this parameter contains the current approximation to the solution vector. Before the first call to the <code>dfgmres</code> routine, it contains the initial approximation to the solution vector.								
<i>b</i>	DOUBLE PRECISION array, this parameter contains the right-hand side vector. Depending on user requests (see the parameter <code>ipar(13)</code>), it might contain the approximate solution after execution.								
<i>RCI_request</i>	<p>INTEGER, this parameter gives information about the result of work of the RCI FGMRES routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:</p> <table border="0" style="margin-left: 2em;"> <tr> <td style="vertical-align: top;"><i>RCI_request</i>= 1</td> <td>multiply the matrix by <code>tmp(ipar(22))</code>, put the result in <code>tmp(ipar(23))</code>, and return the control to the <code>dfgmres</code> routine;</td> </tr> <tr> <td style="vertical-align: top;"><i>RCI_request</i>= 2</td> <td>perform the stopping tests. If they fail, return the control to the <code>dfgres</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;</td> </tr> <tr> <td style="vertical-align: top;"><i>RCI_request</i>= 3</td> <td>apply the preconditioner to <code>tmp(ipar(22))</code>, put the result in <code>tmp(ipar(23))</code>, and return the control to the <code>dfgmres</code> routine.</td> </tr> <tr> <td style="vertical-align: top;"><i>RCI_request</i>= 4</td> <td>check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.</td> </tr> </table>	<i>RCI_request</i> = 1	multiply the matrix by <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine;	<i>RCI_request</i> = 2	perform the stopping tests. If they fail, return the control to the <code>dfgres</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;	<i>RCI_request</i> = 3	apply the preconditioner to <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine.	<i>RCI_request</i> = 4	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.
<i>RCI_request</i> = 1	multiply the matrix by <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine;								
<i>RCI_request</i> = 2	perform the stopping tests. If they fail, return the control to the <code>dfgres</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;								
<i>RCI_request</i> = 3	apply the preconditioner to <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine.								
<i>RCI_request</i> = 4	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.								
<i>ipar(128)</i> [128]	<p>INTEGER array, this parameter specifies the integer set of data for the RCI FGMRES computations:</p> <table border="0" style="margin-left: 2em;"> <tr> <td style="vertical-align: top;"><i>ipar(1)</i></td> <td>specifies the size of the problem. The <code>dfgmres_init</code> routine assigns <code>ipar(1)=n</code>. All the other routines uses this parameter instead of <i>n</i>. There is no default value for this parameter.</td> </tr> </table>	<i>ipar(1)</i>	specifies the size of the problem. The <code>dfgmres_init</code> routine assigns <code>ipar(1)=n</code> . All the other routines uses this parameter instead of <i>n</i> . There is no default value for this parameter.						
<i>ipar(1)</i>	specifies the size of the problem. The <code>dfgmres_init</code> routine assigns <code>ipar(1)=n</code> . All the other routines uses this parameter instead of <i>n</i> . There is no default value for this parameter.								

<i>ipar(2)</i>	specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file <code>MKL_RCI_FGMRES_Log.txt</code> . Note that if <i>ipar(6)</i> and <i>ipar(7)</i> parameters are set to 0, error and warning messages are not generated at all.
<i>ipar(3)</i>	contains the current stage of the RCI FGMRES computations. The initial value is 1.
<hr/> WARNING <hr/>	
Avoid altering this variable during computations.	
<i>ipar(4)</i>	contains the current iteration number. The initial value is 0.
<i>ipar(5)</i>	specifies the maximum number of iterations. The default value is $\min(150, n)$.
<i>ipar(6)</i>	if the value is not 0, the routines output error messages in accordance with the parameter <i>ipar(2)</i> . If it is 0, the routines do not output error messages at all, but return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar(7)</i>	if the value is not 0, the routines output warning messages in accordance with the parameter <i>ipar(2)</i> . Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter <i>RCI_request</i> . The default value is 1.
<i>ipar(8)</i>	if the value is not equal to 0, the <code>dfgmres</code> routine performs the stopping test for the maximum number of iterations: $ipar(4) \leq ipar(5)$. If the value is 0, the <code>dfgmres</code> routine does not perform this stopping test. The default value is 1.
<i>ipar(9)</i>	if the value is not 0, the <code>dfgmres</code> routine performs the residual stopping test: $dpar(5) \leq dpar(4)$. If the value is 0, the <code>dfgmres</code> routine does not perform this stopping test. The default value is 0.
<i>ipar(10)</i>	if the value is not 0, the <code>dfgmres</code> routine indicates that the user-defined stopping test should be performed by setting <i>RCI_request</i> =2. If the value is 0, the <code>dfgmres</code> routine does not perform the user-defined stopping test. The default value is 1.

NOTE

At least one of the parameters $ipar(8)$ - $ipar(10)$ must be set to 1.

- | | |
|------------|---|
| $ipar(11)$ | if the value is 0, the <code>dfgmres</code> routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method, and requests that you perform the preconditioning step by setting the output parameter $RCI_request=3$. The default value is 0. |
| $ipar(12)$ | if the value is not equal to 0, the <code>dfgmres</code> routine performs the automatic test for zero norm of the currently generated vector: $dpar(7) \leq dpar(8)$, where $dpar(8)$ contains the tolerance value. Otherwise, the routine indicates that you must perform this check by setting the output parameter $RCI_request=4$. The default value is 0. |
| $ipar(13)$ | if the value is equal to 0, the <code>dfgmres_get</code> routine updates the solution to the vector x according to the computations done by the <code>dfgmres</code> routine. If the value is positive, the routine writes the solution to the right-hand side vector b . If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0. |

NOTE

It is possible to call the `dfgmres_get` routine at any place in the code, but you must pay special attention to the parameter $ipar(13)$. The RCI FGMRES iterations can be continued after the call to `dfgmres_get` routine only if the parameter $ipar(13)$ is not equal to zero. If $ipar(13)$ is positive, then the updated solution overwrites the right-hand side in the vector b . If you want to run the restarted version of FGMRES with the same right-hand side, then it must be saved in a different memory location before the first call to the `dfgmres_get` routine with positive $ipar(13)$.

- | | |
|------------|--|
| $ipar(14)$ | contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0. |
|------------|--|

WARNING

Do not alter this variable during computations.

<i>ipar(15)</i>	specifies the number of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, assign the number of iterations to <i>ipar(15)</i> before the restart. The default value is $\min(150, n)$, which means that by default the non-restarted version of FGMRES method is used.
<i>ipar(16)</i>	service variable specifying the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see Matrix Arguments in the Appendix B for details) is started in the <i>tmp</i> array.
<i>ipar(17)</i>	service variable specifying the location of the rotation cosines from which the vector of cosines is started in the <i>tmp</i> array.
<i>ipar(18)</i>	service variable specifying the location of the rotation sines from which the vector of sines is started in the <i>tmp</i> array.
<i>ipar(19)</i>	service variable specifying the location of the rotated residual vector from which the vector is started in the <i>tmp</i> array.
<i>ipar(20)</i>	service variable, specifies the location of the least squares solution vector from which the vector is started in the <i>tmp</i> array.
<i>ipar(21)</i>	service variable specifying the location of the set of preconditioned vectors from which the set is started in the <i>tmp</i> array. The memory locations in the <i>tmp</i> array starting from <i>ipar(21)</i> are used only for the preconditioned FGMRES method.
<i>ipar(22)</i>	specifies the memory location from which the first vector (source) used in operations requested via <i>RCI_request</i> is started in the <i>tmp</i> array.
<i>ipar(23)</i>	specifies the memory location from which the second vector (output) used in operations requested via <i>RCI_request</i> is started in the <i>tmp</i> array.
<i>ipar(24:128)</i>	are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array *ipar* with length 128. While defining the array in the code as `INTEGER ipar(23)` works, there is no guarantee of future compatibility with Intel MKL.

dpar(128)

DOUBLE PRECISION array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:

<i>dpar</i> (1)	specifies the relative tolerance. The default value is 1.0e-6.
<i>dpar</i> (2)	specifies the absolute tolerance. The default value is 0.0e-0.
<i>dpar</i> (3)	specifies the Euclidean norm of the initial residual (if it is computed in the <code>dfgmres</code> routine). The initial value is 0.0.
<i>dpar</i> (4)	service variable equal to $dpar(1) * dpar(3) + dpar(2)$ (if it is computed in the <code>dfgmres</code> routine). The initial value is 0.0.
<i>dpar</i> (5)	specifies the Euclidean norm of the current residual. The initial value is 0.0.
<i>dpar</i> (6)	specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.
<i>dpar</i> (7)	contains the norm of the generated vector. The initial value is 0.0.

NOTE

In terms of [Saad03] this parameter is the coefficient $h_{k+1,k}$ of the Hessenberg matrix.

<i>dpar</i> (8)	contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0e-12.
<i>dpar</i> (9:128)	are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array *dpar* with length 128. While defining the array in the code as `DOUBLE PRECISION dpar(8)` works, there is no guarantee of future compatibility with Intel MKL.

<i>tmp</i>	DOUBLE PRECISION array of size $((2 * ipar(15) + 1) * n + ipar(15) * (ipar(15) + 9) / 2 + 1)$ used to supply the double precision temporary space for the RCI FGMRES computations, specifically:
<i>tmp</i> (1:ipar(16)-1)	contains the sequence of vectors generated by the FGMRES method. The initial value is 0.0.
<i>tmp</i> (ipar(16):ipar(17)-1)	contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> (ipar(17):ipar(18)-1)	contains the rotation cosines vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> (ipar(18):ipar(19)-1)	contains the rotation sines vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> (ipar(19):ipar(20)-1)	contains the rotated residual vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> (ipar(20):ipar(21)-1)	contains the solution vector to the least squares problem generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> (ipar(21):)	contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of <i>tmp</i> array is not used if the non-preconditioned version of FGMRES method is called. There is no initial value for this part of <i>tmp</i> array.

NOTE

You can define this array in the code as `DOUBLE PRECISION tmp((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1)` if you run only non-preconditioned FGMRES iterations.

Scheme of Using the RCI FGMRES Routines

The following pseudocode shows the general scheme of using the RCI FGMRES routines.

```

...
generate matrix A
generate preconditioner C (optional)
  call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
  change parameters in ipar, dpar if necessary
  call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1 call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
  if (RCI_request.eq.1) then

```

multiply the matrix A by $tmp(ipar(22))$ and put the result in $tmp(ipar(23))$

It is possible to use [MKL Sparse BLAS Level 2](#) subroutines for this operation

```
c proceed with FGMRES iterations
  goto 1
endif
if (RCI_request.eq.2) then
  do the stopping test
  if (test not passed) then
c proceed with FGMRES iterations
  go to 1
  else
c stop FGMRES iterations
  goto 2
  endif
endif
if (RCI_request.eq.3) then (optional)
  apply the preconditioner  $C$  inverse to  $tmp(ipar(22))$  and put the result in  $tmp(ipar(23))$ 
c proceed with FGMRES iterations
  goto 1
endif
if (RCI_request.eq.4) then
  check the norm of the next orthogonal vector, it is contained in  $dpar(7)$ 
  if (the norm is not zero up to rounding/computational errors) then
c proceed with FGMRES iterations
  goto 1
  else
c stop FGMRES iterations
  goto 2
  endif
endif
2 call dfgmres_get( $n$ ,  $x$ ,  $b$ ,  $RCI\_request$ ,  $ipar$ ,  $dpar$ ,  $tmp$ ,  $itercount$ )
current iteration number is in  $itercount$ 
the computed approximation is in the array  $x$ 
```

NOTE

For the FGMRES method, the array x initially contains the current approximation to the solution. It can be updated only by calling the routine `dfgmres_get`, which updates the solution in accordance with the computations performed by the routine `dfgmres`.

The above pseudocode demonstrates two main differences in the use of RCI FGMRES interface comparing with the [CG Interface Description](#). The first difference relates to `RCI_request=3`: it uses different locations in the `tmp` array, which is two-dimensional for CG and one-dimensional for FGMRES. The second difference relates to `RCI_request=4`: the RCI CG interface never produces `RCI_request=4`.

RCI ISS Routines

`dcg_init`

Initializes the solver.

Syntax

```
dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcg_init` initializes the solver. After initialization, all subsequent invocations of the Intel MKL RCI CG routines use the values of all parameters returned by the routine `dcg_init`. Advanced users can skip this step and set the values in the `ipar` and `dpar` arrays directly.

CAUTION

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	INTEGER. Sets the size of the problem.
x	DOUBLE PRECISION. Array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .
b	DOUBLE PRECISION. Array of size n . Contains the right-hand side vector.

Output Parameters

<code>RCI_request</code>	INTEGER. Gives information about the result of the routine.
<code>ipar</code>	INTEGER . Array of size 128. Refer to the CG Common Parameters .

<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION . Array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dcg_check

Checks consistency and correctness of the user defined data.

Syntax

`dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcg_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcg`. However this operation does not guarantee that the solver returns the correct result. It only reduces the chance of making a mistake in the parameters of the method. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about result of the routine.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

<code>RCI_request= 0</code>	Indicates that the task completed normally.
<code>RCI_request= -1100</code>	Indicates that the task is interrupted and the errors occur.
<code>RCI_request= -1001</code>	Indicates that there are some warning messages.
<code>RCI_request= -1010</code>	Indicates that the routine changed some parameters to make them consistent or correct.
<code>RCI_request= -1011</code>	Indicates that there are some warning messages and that the routine changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

`dcg(n, x, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `dcg` routine computes the approximate solution vector using the CG method [Young71]. The routine `dcg` uses the vector in the array `x` before the first call as an initial approximation to the solution. The parameter `RCI_request` gives you information about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

<code>n</code>	INTEGER. Sets the size of the problem.
<code>x</code>	DOUBLE PRECISION . Array of size <code>n</code> . Contains the initial approximation to the solution vector.
<code>b</code>	DOUBLE PRECISION . Array of size <code>n</code> . Contains the right-hand side vector.
<code>tmp</code>	DOUBLE PRECISION . Array of size <code>(n, 4)</code> . Refer to the CG Common Parameters .

Output Parameters

<code>RCI_request</code>	INTEGER. Gives information about result of work of the routine.
<code>x</code>	DOUBLE PRECISION . Array of size <code>n</code> . Contains the updated approximation to the solution vector.
<code>ipar</code>	INTEGER . Array of size <code>128</code> . Refer to the CG Common Parameters .
<code>dpar</code>	DOUBLE PRECISION . Array of size <code>128</code> . Refer to the CG Common Parameters .

tmp DOUBLE PRECISION . Array of size $(n, 4)$. Refer to the [CG Common Parameters](#).

Return Values

<i>RCI_request</i> =0	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request</i> = 2.
<i>RCI_request</i> =-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
<i>RCI_request</i> =-2	Indicates that the routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
<i>RCI_request</i> =- 10	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value <i>dpar</i> (6) was altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> =-11	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values <i>ipar</i> (8), <i>ipar</i> (9), <i>ipar</i> (10) were altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> = 1	Indicates that you must multiply the matrix by <i>tmp</i> (1:n, 1), put the result in the <i>tmp</i> (1:n,2), and return control back to the routine <i>dcg</i> .
<i>RCI_request</i> = 2	Indicates that you must perform the stopping tests. If they fail, return control back to the <i>dcg</i> routine. Otherwise, the solution is found and stored in the vector <i>x</i> .
<i>RCI_request</i> = 3	Indicates that you must apply the preconditioner to <i>tmp</i> (:, 3), put the result in the <i>tmp</i> (:, 4), and return control back to the routine <i>dcg</i> .

dcg_get

Retrieves the number of the current iteration.

Syntax

dcg_get(*n*, *x*, *b*, *RCI_request*, *ipar*, *dpar*, *tmp*, *itercount*)

Include Files

- Fortran: *mkl_rci.fi*, *mkl_rci.f90*

Description

The routine *dcg_get* retrieves the current iteration number of the solutions process.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the approximation vector to the solution.
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.
<i>RCI_request</i>	INTEGER. This parameter is not used.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 4)$. Refer to the CG Common Parameters .

Output Parameters

<i>itercount</i>	INTEGER. Returns the current iteration number.
------------------	--

Return Values

The routine `dcg_get` has no return values.

dcgmrhs_init

Initializes the RCI CG solver with MHRs.

Syntax

```
dcgmrhs_init(n, x, nrhs, b, method, RCI_request, ipar, dpar, tmp)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcgmrhs_init` initializes the solver. After initialization all subsequent invocations of the Intel MKL RCI CG with multiple right-hand sides (MRHS) routines use the values of all parameters that are returned by `dcgmrhs_init`. Advanced users may skip this step and set the values to these parameters directly in the appropriate routines.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcgmrhs_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
----------	--

<i>x</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	INTEGER. Sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side vectors.
<i>method</i>	INTEGER. Specifies the method of solution: A value of 1 indicates CG with multiple right-hand sides (default value)

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about the result of the routine.
<i>ipar</i>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, (3+nrhs))$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dcgmrhs_check

Checks consistency and correctness of the user defined data.

Syntax

`dcgmrhs_check(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcgmrhs_check` checks the consistency and correctness of the parameters to be passed to the solver routine `dcgmrhs`. While this operation reduces the chance of making a mistake in the parameters, it does not guarantee that the solver returns the correct result.

If you are sure that the correct data is *specified* in the solver parameters, you can skip this operation.

The lengths of all vectors must be defined in a previous call to the `dcgmrhs_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
----------	--

<i>x</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	INTEGER. This parameter sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side vectors.

Output Parameters

<i>RCI_request</i>	INTEGER. Returns information about the results of the routine.
<i>ipar</i>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, (3+nrhs))$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -1100	Indicates that the task is interrupted and the errors occur.
<i>RCI_request</i> = -1001	Indicates that there are some warning messages.
<i>RCI_request</i> = -1010	Indicates that the routine changed some parameters to make them consistent or correct.
<i>RCI_request</i> = -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dcgmrhs

Computes the approximate solution vectors.

Syntax

`dcgmrhs(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcgmrhs` computes approximate solution vectors using the CG with multiple right-hand sides (MRHS) method [Young71]. The routine `dcgmrhs` uses the value that was in the *x* before the first call as an initial approximation to the solution. The parameter *RCI_request* gives information about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcgmrhs_init` routine.

Input Parameters

n	INTEGER. Sets the size of the problem, and the sizes of arrays x and b .
x	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors.
$nrhs$	INTEGER. Sets the number of right-hand sides.
b	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side vectors.
tmp	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Output Parameters

$RCI_request$	INTEGER. Gives information about result of work of the routine.
x	DOUBLE PRECISION. Array of size $(n$ by $nrhs)$. Contains the updated approximation to the solution vectors.
$ipar$	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
$dpar$	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
tmp	DOUBLE PRECISION. Array of size $(n, (3+nrhs))$. Refer to the CG Common Parameters .

Return Values

$RCI_request=0$	Indicates that the task completed normally and the solution is found and stored in the vector x . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the $RCI_request= 2$.
$RCI_request=-1$	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if both tests are requested by the user.
$RCI_request=-2$	The routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
$RCI_request=- 10$	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value $dpar(6)$ was altered outside of the routine, or the <code>dcg_check</code> routine was not called.

<code>RCI_request=-11</code>	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values <code>ipar(8)</code> , <code>ipar(9)</code> , <code>ipar(10)</code> were altered outside of the routine, or the <code>dcg_check</code> routine was not called.
<code>RCI_request= 1</code>	Indicates that you must multiply the matrix by <code>tmp(1:n, 1)</code> , put the result in the <code>tmp(1:n,2)</code> , and return control back to the routine <code>dcg</code> .
<code>RCI_request= 2</code>	Indicates that you must perform the stopping tests. If they fail, return control back to the <code>dcg</code> routine. Otherwise, the solution is found and stored in the vector <code>x</code> .
<code>RCI_request= 3</code>	Indicates that you must apply the preconditioner to <code>tmp(:, 3)</code> , put the result in the <code>tmp(:, 4)</code> , and return control back to the routine <code>dcg</code> .

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

```
dcgmrhs_get(n, x, nrhs, b, RCI_request, ipar, dpar, tmp, itercount)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcgmrhs_get` retrieves the current iteration number of the solving process.

Input Parameters

<code>n</code>	INTEGER. Sets the size of the problem.
<code>x</code>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors.
<code>nrhs</code>	INTEGER. Sets the number of right-hand sides.
<code>b</code>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side .
<code>RCI_request</code>	INTEGER. This parameter is not used.
<code>ipar</code>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<code>dpar</code>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<code>tmp</code>	DOUBLE PRECISION. Array of size $(n, (3+nrhs))$. Refer to the CG Common Parameters .

Output Parameters

itercount INTEGER. Array of size *nrhs*. Returns the current iteration number for each right-hand side.

Return Values

The routine `dcgmrhs_get` has no return values.

dfgmres_init

Initializes the solver.

Syntax

```
dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres_init` initializes the solver. After initialization all subsequent invocations of Intel MKL RCI FGMRES routines use the values of all parameters that are returned by `dfgmres_init`. Advanced users can skip this step and set the values in the *ipar* and *dpar* arrays directly.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n INTEGER. Sets the size of the problem.

x DOUBLE PRECISION. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.

b DOUBLE PRECISION. Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request INTEGER. Gives information about the result of the routine.

ipar INTEGER. Array of size 128. Refer to the [FGMRES Common Parameters](#).

dpar DOUBLE PRECISION. Array of size 128. Refer to the [FGMRES Common Parameters](#).

tmp DOUBLE PRECISION. Array of size $((2 * ipar(15) + 1) * n + ipar(15) * (ipar(15) + 9) / 2 + 1)$. Refer to the [FGMRES Common Parameters](#).

Return Values

<code>RCI_request= 0</code>	Indicates that the task completed normally.
<code>RCI_request= -10000</code>	Indicates failure to complete the task.

dfgmres_check

Checks consistency and correctness of the user defined data.

Syntax

```
dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres_check` checks consistency and correctness of the parameters to be passed to the solver routine `dfgmres`. However, this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the routine. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors are assumed to have been defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<code>n</code>	INTEGER. Sets the size of the problem.
<code>x</code>	DOUBLE PRECISION. Array of size <code>n</code> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <code>b</code> .
<code>b</code>	DOUBLE PRECISION. Array of size <code>n</code> . Contains the right-hand side vector.

Output Parameters

<code>RCI_request</code>	INTEGER. Gives information about result of the routine.
<code>ipar</code>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<code>dpar</code>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<code>tmp</code>	DOUBLE PRECISION. Array of size $((2*ipar(15) + 1)*n + ipar(15)*(ipar(15) + 9)/2 + 1)$. Refer to the FGMRES Common Parameters .

Return Values

<code>RCI_request= 0</code>	Indicates that the task completed normally.
<code>RCI_request= -1100</code>	Indicates that the task is interrupted and the errors occur.
<code>RCI_request= -1001</code>	Indicates that there are some warning messages.

<code>RCI_request= -1010</code>	Indicates that the routine changed some parameters to make them consistent or correct.
<code>RCI_request= -1011</code>	Indicates that there are some warning messages and that the routine changed some parameters.

dfgmres

Makes the FGMRES iterations.

Syntax

```
dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres` performs the FGMRES iterations [Saad03], using the value that was in the array x before the first call as an initial approximation of the solution vector. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter `ipar(13)` is not equal to 0 (default value). Note that the updated solution overwrites the right-hand side in the vector b if the parameter `ipar(13)` is positive, and the restarted version of the FGMRES method can not be run. If you want to keep the right-hand side, you must save it in a different memory location before the first call to the `dfgmres_get` routine with a positive `ipar(13)`.

The parameter `RCI_request` gives information about the task completion and requests results of certain operations that the solver requires.

The lengths of all the vectors must be defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<code>n</code>	INTEGER. Sets the size of the problem.
<code>x</code>	DOUBLE PRECISION. Array of size n . Contains the initial approximation to the solution vector.
<code>b</code>	DOUBLE PRECISION. Array of size n . Contains the right-hand side vector.
<code>tmp</code>	DOUBLE PRECISION. Array of size (13). Refer to the FGMRES Common Parameters .

Output Parameters

<code>RCI_request</code>	INTEGER. Informs about result of work of the routine.
<code>ipar</code>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<code>dpar</code>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .

tmp DOUBLE PRECISION. Array of size $((2 * ipar(15) + 1) * n + ipar(15) * ipar(15) + 9) / 2 + 1$. Refer to the [FGMRES Common Parameters](#).

Return Values

<i>RCI_request</i> =0	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request</i> = 2 or 4.
<i>RCI_request</i> =-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
<i>RCI_request</i> = -10	Indicates that the routine was interrupted because of an attempt to divide by zero. Usually this happens if the matrix is degenerate or almost degenerate. However, it may happen if the parameter <i>dpar</i> is altered, or if the method is not stopped when the solution is found.
<i>RCI_request</i> = -11	Indicates that the routine was interrupted because it entered an infinite cycle. Usually this happens because the values <i>ipar</i> (8), <i>ipar</i> (9), <i>ipar</i> (10) were altered outside of the routine, or the <i>dfgmres_check</i> routine was not called.
<i>RCI_request</i> = -12	Indicates that the routine was interrupted because errors were found in the method parameters. Usually this happens if the parameters <i>ipar</i> and <i>dpar</i> were altered by mistake outside the routine.
<i>RCI_request</i> = 1	Indicates that you must multiply the matrix by <i>tmp</i> (<i>ipar</i> (22)), put the result in the <i>tmp</i> (<i>ipar</i> (23)), and return control back to the routine <i>dfgmres</i> .
<i>RCI_request</i> = 2	Indicates that you must perform the stopping tests. If they fail, return control to the <i>dfgmres</i> routine. Otherwise, the FGMRES solution is found, and you can run the <i>fgmres_get</i> routine to update the computed solution in the vector <i>x</i> .
<i>RCI_request</i> = 3	Indicates that you must apply the inverse preconditioner to <i>ipar</i> (22), put the result in the <i>ipar</i> (23), and return control back to the routine <i>dfgmres</i> .
<i>RCI_request</i> = 4	Indicates that you must check the norm of the currently generated vector. If it is not zero within the computational/rounding errors, return control to the <i>dfgmres</i> routine. Otherwise, the FGMRES solution is found, and you can run the <i>dfgmres_get</i> routine to update the computed solution in the vector <i>x</i> .

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

```
dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dfgmres_get` retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the `dfgmres` routine. To retrieve the current iteration number only, set the parameter `ipar(13) = -1` beforehand. Normally, you should do this before proceeding further with the computations. If the intermediate solution is needed, the method parameters must be set properly. For details see [FGMRES Common Parameters](#) and the Iterative Sparse Solver code examples in the Intel MKL installation directory:

- `examples/solverf/source`

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $((2 * ipar(15) + 1) * n + ipar(15) * ipar(15) + 9) / 2 + 1$. Refer to the FGMRES Common Parameters .

Output Parameters

<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . If <code>ipar(13) = 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . If <code>ipar(13) > 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>RCI_request</i>	INTEGER. Gives information about result of the routine.
<i>itercount</i>	INTEGER. Contains the value of the current iteration number.

Return Values

`RCI_request = 0` Indicates that the task completed normally.

`RCI_request= -12`

Indicates that the routine was interrupted because errors were found in the routine parameters. Usually this happens if the parameters `ipar` and `dpar` were altered by mistake outside of the routine.

`RCI_request= -10000`

Indicates that the routine failed to complete the task.

Implementation Details

Several aspects of the Intel MKL RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel MKL RCI ISS language-specific header files.

NOTE

Intel MKL does not support the RCI ISS interface unless you include the language-specific header file.

Preconditioners based on Incomplete LU Factorization Technique

Preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms *preconditioner* and *accelerator* are synonyms, hereafter only *preconditioner* is used.

Intel MKL provides two preconditioners, ILU0 and ILUT, for sparse matrices presented in the format accepted in the Intel MKL direct sparse solvers (three-array variation of the CSR storage format described in [Sparse Matrix Storage Format](#)). The algorithms used are described in [[Saad03](#)].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: lower and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT algorithm is that it calculates each element of the preconditioner and saves each one if it satisfies two conditions simultaneously: its value is greater than the product of the given tolerance and matrix row norm, and its value is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel MKL RCI FGMRES solver (see [Sparse Solver Routines](#)). Avoid using these preconditioners with MKL RCI CG solver because in general, they produce a non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel MKL triangular solver routine `mkl_dcsrtrsv` must be applied twice: for the lower triangular part of the preconditioner, and then for its upper triangular part.

NOTE

Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. Whether or not the preconditioner produces an acceptable result can only be determined in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial solution, and even ruin the convergence. It is your responsibility as a user to choose a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The general scheme for use is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. You can see all these differences in the preconditioner code examples (`dcsrilu*.*`) in the `examples` folder of the Intel MKL installation directory:

- `examples/solverf/source`

The following pseudocode shows the general scheme of using the ILUT preconditioner in the RCI FGMRES context.

...

generate matrix *A*

generate preconditioner *C* (optional)

```
call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

```
change parameters in ipar, dpar if necessary
```

```
call dcsrilit(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
```

```
call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
```

```
1 call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
```

```
if (RCI_request.eq.1) then
```

```
multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
```

```
c proceed with FGMRES iterations
```

```
goto 1
```

```
endif
```

```
if (RCI_request.eq.2) then
```

```
do the stopping test
```

```
if (test not passed) then
```

```
c proceed with FGMRES iterations
```

```
go to 1
```

```
else
```

```
c stop FGMRES iterations.
```

```
goto 2
```

```
endif
```

```
endif
```

```
if (RCI_request.eq.3) then
```

```
c Below, trvec is an intermediate vector of length at least n
```

```
c Here is the recommended use of the result produced by the ILUT routine.
```

```
c via standard Intel MKL Sparse Blas solver routine mkl_dcsrtrsv.
```

```
call mkl_dcsrtrsv('L','N','U', n, bilut, ibilut, jbilut, tmp(ipar(22)), trvec)
```

```
call mkl_dcsrtrsv('U','N','N', n, bilut, ibilut, jbilut, trvec, tmp(ipar(23)))
```

```
c proceed with FGMRES iterations
```

```
goto 1
```

```
endif
if (RCI_request.eq.4) then
  check the norm of the next orthogonal vector, it is contained in dpar(7)
  if (the norm is not zero up to rounding/computational errors) then
c proceed with FGMRES iterations
  goto 1
  else
c stop FGMRES iterations
  goto 2
endif
endif
endif
2 call dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
current iteration number is in itercount
the computed approximation is in the array x
```

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel MKL preconditioner routines are discussed in the [Appendix A Linear Solvers Basics](#).

In this section FORTRAN style notations are used. All types refer to the standard Fortran types, `INTEGER`, and `DOUBLE PRECISION`.

User Data Arrays

The preconditioner routines take arrays of user data as input. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL preconditioner routines do not make copies of the user input arrays.

Common Parameters

Some parameters of the preconditioners are common with the [FGMRES Common Parameters](#). The routine `dfgmres_init` specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:

ipar(2) - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter *ipar*(6) is set to 0, then error messages are not generated at all.

ipar(6) - specifies whether error messages are generated. If its value is not equal to 0, the ILU0 routine returns error messages as specified by the parameter *ipar*(2). Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter *ierr*. The default value is 1.

For the ILUT preconditioner:

ipar(2) - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter *ipar*(6) is set to 0, then error messages are not generated at all.

$ipar(6)$ - specifies whether error messages are generated. If its value is not equal to 0, the ILUT routine returns error messages as specified by the parameter $ipar(2)$. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter $ierr$. The default value is 1.

$ipar(7)$ - if its value is greater than 0, the ILUT routine generates warning messages as specified by the parameter $ipar(2)$ and continues calculations. If its value is equal to 0, the routine returns a positive value of the parameter $ierr$. If its value is less than 0, the routine generates a warning message as specified by the parameter $ipar(2)$ and returns a positive value of the parameter $ierr$. The default value is 1.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix.

Syntax

```
call dcsrilu0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcsrilu0` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L * U$, where L is a lower triangular matrix with a unit diagonal, U is an upper triangular matrix with a non-unit diagonal, and the portrait of the original matrix A is used to store the incomplete factors L and U .

CAUTION

This routine supports only one-based indexing of the array parameters.

Input Parameters

n	INTEGER. Size (number of rows or columns) of the original square n -by- n matrix A .
a	DOUBLE PRECISION. Array containing the set of elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to the <i>values</i> array description in the Sparse Matrix Storage Format for more details.
ia	INTEGER. Array of size $(n+1)$ containing begin indices of rows of the matrix A such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(n+1)$ is equal to the number of non-zero elements in the matrix A , plus one. Refer to the <i>rowIndex</i> array description in the Sparse Matrix Storage Format for more details.
ja	INTEGER. Array containing the column indices for each non-zero element of the matrix A . It is important that the indices are in increasing order per row. The matrix size is equal to the size of the array a . Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format for more details.

CAUTION

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

ipar

INTEGER. Array of size 128. This parameter specifies the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the *ipar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

ipar(31) specifies how the routine operates when a zero diagonal element occurs during calculation. If this parameter is set to 0 (the default value set by the routine `dfgmres_init`), then the calculations are stopped and the routine returns a non-zero error value. Otherwise, the diagonal element is set to the value of *dpar*(32) and the calculations continue.

NOTE

You can declare the *ipar* array with a size of 32. However, for future compatibility you must declare the array *ipar* with length 128.

dpar

DOUBLE PRECISION. Array of size 128. This parameter specifies the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the *dpar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries specific to ILU0 are listed below.

dpar(31) specifies a small value, which is compared with the computed diagonal elements. When *ipar*(31) is not 0, then diagonal elements less than *dpar*(31) are set to *dpar*(32). The default value is 1.0e-16.

NOTE

This parameter can be set to the negative value, because the calculation uses its absolute value.
If this parameter is set to 0, the comparison with the diagonal element is not performed.

dpar(32) specifies the value that is assigned to the diagonal element if its value is less than *dpar*(31) (see above). The default value is 1.0e-10.

NOTE

You can declare the *dpar* array with a size of 32. However, for future compatibility you must declare the array *dpar* with length 128.

Output Parameters

<i>bilu0</i>	DOUBLE PRECISION. Array <i>B</i> containing non-zero elements of the resulting preconditioning matrix <i>B</i> , stored in the format accepted in direct sparse solvers. Its size is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to the <i>values</i> array description in the Sparse Matrix Storage Format section for more details.
<i>ierr</i>	INTEGER. Error flag, gives information about the routine completion.

NOTE

To present the resulting preconditioning matrix in the CSR3 format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Return Values

<i>ierr</i> =0	Indicates that the task completed normally.
<i>ierr</i> =-101	Indicates that the routine was interrupted and that error occurred: at least one diagonal element is omitted from the matrix in CSR3 format (see Sparse Matrix Storage Format).
<i>ierr</i> =-102	Indicates that the routine was interrupted because the matrix contains a diagonal element with the value of zero.
<i>ierr</i> =-103	Indicates that the routine was interrupted because the matrix contains a diagonal element which is so small that it could cause an overflow, or that it would cause a bad approximation to ILU0.
<i>ierr</i> =-104	Indicates that the routine was interrupted because the memory is insufficient for the internal work array.
<i>ierr</i> =-105	Indicates that the routine was interrupted because the input matrix size <i>n</i> is less than or equal to 0.
<i>ierr</i> =-106	Indicates that the routine was interrupted because the column indices <i>ja</i> are not in the ascending order.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilu0 (n, a, ia, ja, bilu0, ipar, dpar, ierr)
INTEGER n, ierr, ipar(128)
INTEGER ia(*), ja(*)
DOUBLE PRECISION a(*), bilu0(*), dpar(128)
```

dcsrilit

ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax

```
call dcsrilit(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine `dcsrilit` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L * U$, where L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors L and U :

- 1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter `tol`, and
- 2) the number of the non-zero elements in each row of the resulting L and U factors must not be greater than the value of the parameter `maxfil`.

CAUTION

This routine supports only one-based indexing of the array parameters.

Input Parameters

<code>n</code>	INTEGER. Size (number of rows or columns) of the original square n -by- n matrix A .
<code>a</code>	DOUBLE PRECISION. Array containing all non-zero elements of the matrix A . The length of the array is equal to their number. Refer to <code>values</code> array description in the Sparse Matrix Storage Format section for more details.
<code>ia</code>	INTEGER. Array of size $(n+1)$ containing indices of non-zero elements in the array <code>a</code> . <code>ia(i)</code> is the index of the first non-zero element from the row i . The value of the last element <code>ia(n+1)</code> is equal to the number of non-zeros in the matrix A , plus one. Refer to the <code>rowIndex</code> array description in the Sparse Matrix Storage Format for more details.
<code>ja</code>	INTEGER. Array of size equal to the size of the array <code>a</code> . This array contains the column numbers for each non-zero element of the matrix A . It is important that the indices are in increasing order per row. Refer to the <code>columns</code> array description in the Sparse Matrix Storage Format for more details.

CAUTION

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

<code>tol</code>	DOUBLE PRECISION. Tolerance for threshold criterion for the resulting entries of the preconditioner.
<code>maxfil</code>	INTEGER. Maximum fill-in, which is half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner cannot exceed $(2 * \text{maxfil} + 1)$.

ipar

INTEGER. Array of size 128. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the *ipar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

ipar(31)

specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter *tol*. If *ipar*(31) = 0, then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to a value determined by *dpar*(31) (see its description below), and the calculations continue.

NOTE

There is no default value for *ipar*(31) even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array *ipar* with length 128. While defining the array in the code as `INTEGER ipar(31)` works, there is no guarantee of future compatibility with Intel MKL.

dpar

DOUBLE PRECISION. Array of size 128. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the *dpar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.

dpar(31)

used to adjust the value of small diagonal elements. Diagonal elements with a value less than the current matrix row norm multiplied by *tol* are replaced with the value of *dpar*(31) multiplied by the matrix row norm.

NOTE

There is no default value for *dpar*(31) entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array *dpar* with length 128. While defining the array in the code as `DOUBLE PRECISION ipar(31)` works, there is no guarantee of future compatibility with Intel MKL.

Output Parameters

bilut DOUBLE PRECISION. Array containing non-zero elements of the resulting preconditioning matrix B , stored in the format accepted in the direct sparse solvers. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) for more details. The size of the array is equal to $(2*maxfil+1)*n-maxfil*(maxfil+1)+1$.

NOTE

Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

ibilut INTEGER. Array of size $(n+1)$ containing indices of non-zero elements in the array *bilut*. *ibilut*(i) is the index of the first non-zero element from the row i . The value of the last element *ibilut*($n+1$) is equal to the number of non-zeros in the matrix B , plus one. Refer to the *rowIndex* array description in the [Sparse Matrix Storage Format](#) for more details.

jbilut INTEGER. Array, its size is equal to the size of the array *bilut*. This array contains the column numbers for each non-zero element of the matrix B . Refer to the *columns* array description in the [Sparse Matrix Storage Format](#) for more details.

ierr INTEGER. Error flag, gives information about the routine completion.

Return Values

<i>ierr</i> =0	Indicates that the task completed normally.
<i>ierr</i> =-101	Indicates that the routine was interrupted because of an error: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.
<i>ierr</i> =-102	Indicates that the routine was interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as <i>ipar</i> (31)=0.
<i>ierr</i> =-103	Indicates that the routine was interrupted because the element <i>ia</i> ($i + 1$) is less than or equal to the element <i>ia</i> (i) (see Sparse Matrix Storage Format).
<i>ierr</i> =-104	Indicates that the routine was interrupted because the memory is insufficient for the internal work arrays.
<i>ierr</i> =-105	Indicates that the routine was interrupted because the input value of <i>maxfil</i> is less than 0.
<i>ierr</i> =-106	Indicates that the routine was interrupted because the size n of the input matrix is less than 0.

<code>ierr=-107</code>	Indicates that the routine was interrupted because an element of the array <code>ja</code> is less than 1, or greater than <code>n</code> (see Sparse Matrix Storage Format).
<code>ierr=101</code>	The value of <code>maxfil</code> is greater than or equal to <code>n</code> . The calculation is performed with the value of <code>maxfil</code> set to <code>(n-1)</code> .
<code>ierr=102</code>	The value of <code>tol</code> is less than 0. The calculation is performed with the value of the parameter set to <code>(-tol)</code> .
<code>ierr=103</code>	The absolute value of <code>tol</code> is greater than value of <code>dpar(31)</code> ; it can result in instability of the calculation.
<code>ierr=104</code>	The value of <code>dpar(31)</code> is equal to 0. It can cause calculations to fail.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilit (n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
INTEGER n, ierr, ipar(*), maxfil
INTEGER ia(*), ja(*), ibilut(*), jbilut(*)
DOUBLE PRECISION a(*), bilut(*), dpar(*), tol
```

Sparse Matrix Checker Routines

Intel MKL provides a sparse matrix checker so that you can find errors in the storage of sparse matrices before calling Intel MKL PARDISO, DSS, or Sparse BLAS routines.

`sparse_matrix_checker`

Checks correctness of sparse matrix.

Syntax

```
error = sparse_matrix_checker (handle)
```

Include Files

- `mkl_sparse_handle.fi`, `mkl_sparse_handle.f90`

Description

The `sparse_matrix_checker` routine checks a user-defined array used to store a sparse matrix in order to detect issues which could cause problems in routines that require sparse input matrices, such as Intel MKL PARDISO, DSS, or Sparse BLAS.

Input Parameters

<code>handle</code>	TYPE (SPARSE_STRUCT), INTENT(INOUT) Pointer to the data structure describing the sparse array to check.
---------------------	--

Return Values

The routine returns a value *error*. Additionally, the *check_result* parameter returns information about where the error occurred, which can be used when *message_level* is MKL_NO_PRINT.

Sparse Matrix Checker Error Values

<i>error</i> value	Meaning	Location
MKL_SPARSE_CHECKER_SUCCESS	The input array successfully passed all checks.	
MKL_SPARSE_CHECKER_NON_MONOTONIC	The input array is not 0 or 1 based (<i>ia</i> (1), is not 0 or 1) or elements of <i>ia</i> are not in non-decreasing order as required.	<p>Fortran:</p> <p><i>ia</i>(<i>i</i> + 1) and <i>ia</i>(<i>i</i> + 2) are incompatible.</p> <p><i>check_result</i>(1) = <i>i</i></p> <p><i>check_result</i>(2) = <i>ia</i>(<i>i</i> + 1)</p> <p><i>check_result</i>(3) = <i>ia</i>(<i>i</i> + 2)</p>
MKL_SPARSE_CHECKER_OUT_OF_RANGE	The value of the <i>ja</i> array is lower than the number of the first column or greater than the number of the last column.	<p>Fortran:</p> <p><i>ia</i>(<i>i</i> + 1) and <i>ia</i>(<i>i</i> + 2) are incompatible.</p> <p><i>check_result</i>(1) = <i>i</i></p> <p><i>check_result</i>(2) = <i>ia</i>(<i>i</i> + 1)</p> <p><i>check_result</i>(3) = <i>ia</i>(<i>i</i> + 2)</p>
MKL_SPARSE_CHECKER_NON_TRIANGULAR	The <i>matrix_structure</i> parameter is MKL_UPPER_TRIANGULAR and both <i>ia</i> and <i>ja</i> are not upper triangular, or the <i>matrix_structure</i> parameter is MKL_LOWER_TRIANGULAR and both <i>ia</i> and <i>ja</i> are not lower triangular	<p>Fortran:</p> <p><i>ia</i>(<i>i</i> + 1) and <i>ja</i>(<i>j</i> + 1) are incompatible.</p> <p><i>check_result</i>(1) = <i>i</i></p> <p><i>check_result</i>(2) = <i>ia</i>(<i>i</i> + 1) = <i>j</i></p> <p><i>check_result</i>(3) = <i>ja</i>(<i>j</i> + 1)</p>
MKL_SPARSE_CHECKER_NON_ORDERED	The elements of the <i>ja</i> array are not in non-decreasing order in each row as required.	<p>Fortran:</p> <p><i>ja</i>(<i>j</i> + 1) and <i>ja</i>(<i>j</i> + 2) are incompatible.</p> <p><i>check_result</i>(1) = <i>j</i></p> <p><i>check_result</i>(2) = <i>ja</i>(<i>j</i> + 1)</p> <p><i>check_result</i>(3) = <i>ja</i>(<i>j</i> + 2)</p>

See Also

[sparse_matrix_init](#) Initializes handle for sparse matrix checker.
[Intel MKL PARDISO - Parallel Direct Sparse Solver Interface](#)
[Sparse BLAS Level 2 and Level 3 Routines](#)
[Sparse Matrix Storage Formats](#)

sparse_matrix_init

Initializes handle for sparse matrix checker.

Syntax

call sparse_matrix_init (*handle*)

Include Files

- mkl_sparse_handle.fi, mkl_sparse_handle.f90

Description

The `sparse_matrix_init` routine initializes the handle for the `sparse_matrix_checker` routine. The `handle` variable contains this data:

Description of `sparse_matrix_checker` *handle* Data

Field	Type	Possible Values	Meaning
<code>n</code>	INTEGER		Order of the matrix stored in sparse array.
<code>csr_ia</code>	INTEGER (C_INTPTR_T)	Pointer to <i>ia</i> array for <code>matrix_format =</code> MKL_CSR	
<code>csr_ja</code>	INTEGER (C_INTPTR_T)	Pointer to <i>ja</i> array for <code>matrix_format =</code> MKL_CSR	
<code>check_result(3)</code>	INTEGER (KIND=4)	See Sparse Matrix Checker Error Values for a description of the values returned in <code>check_result</code> .	Indicates location of problem in array when <code>message_level =</code> MKL_NO_PRINT.
<code>indexing</code>	INTEGER (KIND=4)	MKL_ZERO_BASED MKL_ONE_BASED	Indexing style used in array.
<code>matrix_structure</code>	INTEGER (KIND=4)	MKL_GENERAL_STRUCTURE MKL_UPPER_TRIANGULAR MKL_LOWER_TRIANGULAR MKL_STRUCTURAL_SYMMETRIC	Type of sparse matrix stored in array.
<code>matrix_format</code>	INTEGER (KIND=4)	MKL_CSR	Format of array used for sparse matrix storage.
<code>message_level</code>	INTEGER (KIND=4)	MKL_NO_PRINT MKL_PRINT	Determines whether or not feedback is provided on the screen.

Extended Eigensolver Routines

The Extended Eigensolver functionality in Intel Math Kernel Library (Intel MKL) is based on the FEAST Eigenvalue Solver 2.0 (<http://www.ecs.umass.edu/~polizzi/feast>) in compliance with a BSD license agreement.

Extended Eigensolver uses the same naming convention as the original FEAST package.

NOTE

Intel MKL only supports the shared memory programming (SMP) version of the eigenvalue solver.

- [The FEAST Algorithm](#) gives a brief description of the algorithm underlying the Extended Eigensolver.
- [Extended Eigensolver Functionality](#) describes the problems that can and cannot be solved with the Extended Eigensolver and how to get the best results from the routines.
- [Extended Eigensolver Interfaces](#) gives a reference for calling Extended Eigensolver routines.

The FEAST Algorithm

The Extended Eigensolver functionality is a set of high-performance numerical routines for solving symmetric standard eigenvalue problems, $Ax=\lambda x$, or generalized symmetric-definite eigenvalue problems, $Ax=\lambda Bx$. It yields all the eigenvalues (λ) and eigenvectors (x) within a given search interval $[\lambda_{\min}, \lambda_{\max}]$. It is based on the FEAST algorithm, an innovative fast and stable numerical algorithm presented in [\[Polizzi09\]](#), which fundamentally differs from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [\[Bai00\]](#)) or other Davidson-Jacobi techniques [\[Sleijpen96\]](#). The FEAST algorithm is inspired by the density-matrix representation and contour integration techniques in quantum mechanics.

The FEAST numerical algorithm obtains eigenpair solutions using a numerically efficient contour integration technique. The main computational tasks in the FEAST algorithm consist of solving a few independent linear systems along the contour and solving a reduced eigenvalue problem. Consider a circle centered in the middle of the search interval $[\lambda_{\min}, \lambda_{\max}]$. The numerical integration over the circle in the current version of FEAST is performed using N_e -point Gauss-Legendre quadrature with x_e the e -th Gauss node associated with the weight ω_e . For example, for the case $N_e = 8$:

$$\begin{aligned} (x_1, \omega_1) &= (0.183434642495649, 0.362683783378361), \\ (x_2, \omega_2) &= (-0.183434642495649, 0.362683783378361), \\ (x_3, \omega_3) &= (0.525532409916328, 0.313706645877887), \\ (x_4, \omega_4) &= (-0.525532409916328, 0.313706645877887), \\ (x_5, \omega_5) &= (0.796666477413626, 0.222381034453374), \\ (x_6, \omega_6) &= (-0.796666477413626, 0.222381034453374), \\ (x_7, \omega_7) &= (0.960289856497536, 0.101228536290376), \text{ and} \\ (x_8, \omega_8) &= (-0.960289856497536, 0.101228536290376). \end{aligned}$$

The figure [FEAST Pseudocode](#) shows the basic pseudocode for the FEAST algorithm for the case of real symmetric (left pane) and complex Hermitian (right pane) generalized eigenvalue problems, using N for the size of the system and M for the number of eigenvalues in the search interval (see [\[Polizzi09\]](#)).

NOTE

The pseudocode presents a simplified version of the actual algorithm. Refer to <http://arxiv.org/abs/1302.0432> for an in-depth presentation and mathematical proof of convergence of FEAST.

FEAST Pseudocode

<p>A: real symmetric B: symmetric positive definite (SPD) $\Re\{x\}$: real part of x</p> <ol style="list-style-type: none"> 1. Select $M_0 > M$ random vectors $Y_{N \times M_0} \in \mathbb{R}^{N \times M_0}$. 2. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$; For $e = 1, \dots, N_e$ compute $\theta_e = -(\pi / 2)(x_e - 1)$, compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$, solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N \times M_0}$ compute $Q = Q - (\omega_e / 2)\Re\{r \exp(i\theta_e)Q_e\}$ End 3. Form $A_{Q_{M_0 \times M_0}} = Q^T A Q$ and $B_{Q_{M_0 \times M_0}} = Q^T B Q$ reduce value of M_0 if B_Q is not symmetric positive definite. 4. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{R}^{M_0 \times M_0}$. 5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$. If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue solution and its eigenvector is X_m (the m-th column of X). 6. Check convergence for the trace of the eigenvalues λ_m. If iterative refinement is needed, compute $Y = BX$ and go back to step 2. 	<p>A: complex Hermitian B: Hermitian positive definite (HPD)</p> <ol style="list-style-type: none"> 1. Select $M_0 > M$ random vectors $Y_{N \times M_0} \in \mathbb{C}^{N \times M_0}$. 2. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$; For $e = 1, \dots, N_e$ compute $\theta_e = -(\pi / 2)(x_e - 1)$, compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$, solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N \times M_0}$ solve $(Z_e B - A)^H \hat{Q}_e = Y$ to obtain $\hat{Q}_e \in \mathbb{C}^{N \times M_0}$ $Q = Q - (\omega_e / 4)r(\exp(i\theta_e)Q_e + \exp(-i\theta_e)\hat{Q}_e)$ End 3. Form $A_{Q_{M_0 \times M_0}} = Q^H A Q$ and $B_{Q_{M_0 \times M_0}} = Q^H B Q$ reduce value of M_0 if B_Q is not Hermitian positive definite. 4. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{C}^{M_0 \times M_0}$. 5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$. If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue solution and its eigenvector is X_m (the m-th column of X). 6. Check convergence for the trace of the eigenvalues λ_m. If iterative refinement is needed, compute $Y = BX$ and go back to step 2.
---	---

Extended Eigensolver Functionality

Use Extended Eigensolver to compute all the eigenvalues and eigenvectors within a given search interval.

The eigenvalue problems covered are as follows:

- standard, $Ax = \lambda x$
 - A complex Hermitian
 - A real symmetric
- generalized, $Ax = \lambda Bx$
 - A complex Hermitian, B Hermitian positive definite (hpd)
 - A real symmetric and B real symmetric positive definite (spd)

The Extended Eigensolver functionality offers:

- Real/Complex and Single/Double precisions: double precision is recommended to provide better accuracy of eigenpairs.
- Reverse communication interfaces (RCI) provide maximum flexibility for specific applications. RCI are independent of matrix format and inner system solvers, so you must provide your own linear system solvers (direct or iterative) and matrix-matrix multiply routines.
- Predefined driver interfaces for dense, LAPACK banded, and sparse (CSR) formats are less flexible but are optimized and easy to use:
 - The Extended Eigensolver interfaces for dense matrices are likely to be slower than the comparable LAPACK routines because the FEAST algorithm has a higher computational cost.

- The Extended Eigensolver interfaces for banded matrices support banded LAPACK-type storage.
- The Extended Eigensolver sparse interfaces support compressed sparse row format and use the Intel MKL PARDISO solver.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Parallelism in Extended Eigensolver Routines

How you achieve parallelism in Extended Eigensolver routines depends on which interface you use. Parallelism (via shared memory programming) is not *explicitly* implemented in Extended Eigensolver routines within one node: the inner linear systems are currently solved one after another.

- Using the Extended Eigensolver RCI interfaces, you can achieve parallelism by providing a threaded inner system solver and a matrix-matrix multiplication routine. When using the RCI interfaces, you are responsible for activating the threaded capabilities of your BLAS and LAPACK libraries most likely using the shell variable `OMP_NUM_THREADS`.
- Using the predefined Extended Eigensolver interfaces, parallelism can be implicitly obtained within the shared memory version of BLAS, LAPACK or Intel MKL PARDISO. The shell variable `MKL_NUM_THREADS` can be used for automatically setting the number of OpenMP threads (cores) for BLAS, LAPACK, and Intel MKL PARDISO.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Achieving Performance With Extended Eigensolver Routines

In order to use the Extended Eigensolver Routines, you need to provide

- the search interval and the size of the subspace M_0 (overestimation of the number of eigenvalues M within a given search interval);
- the system matrix in dense, banded, or sparse CSR format if the Extended Eigensolver predefined interfaces are used, or a high-performance complex direct or iterative system solver and matrix-vector multiplication routine if RCI interfaces are used.

In return, you can expect

- fast convergence with very high accuracy when seeking up to 1000 eigenpairs (in two to four iterations using $M_0 = 1.5M$, and $N_e = 8$ or at most using $N_e = 16$ contour points);
- an extremely robust approach.

The performance of the basic FEAST algorithm depends on a trade-off between the choices of the number of Gauss quadrature points N_e , the size of the subspace M_0 , and the number of outer refinement loops to reach the desired accuracy. In practice you should use $M_0 > 1.5 M$, $N_e = 8$, and at most two refinement loops.

For better performance:

- M_0 should be much smaller than the size of the eigenvalue problem, so that the arithmetic complexity depends mainly on the inner system solver ($O(NM)$ for narrow-banded or sparse systems).
- Parallel scalability performance depends on the shared memory capabilities of the of the inner system solver.
- For very large sparse and challenging systems, application users should make use of the Extended Eigensolver RCI interfaces with customized highly-efficient iterative systems solvers and preconditioners.
- For the Extended Eigensolver interfaces for banded matrices, the parallel performance scalability is limited.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Extended Eigensolver Interfaces

Extended Eigensolver Naming Conventions

There are two different types of interfaces available in the Extended Eigensolver routines:

1. The reverse communication interfaces (RCI):

```
?feast_<matrix type>_rci
```

These interfaces are matrix free format (the interfaces are independent of the matrix data formats). You must provide matrix-vector multiply and direct/iterative linear system solvers for your own explicit or implicit data format.

2. The predefined interfaces:

```
?feast_<matrix type><type of eigenvalue problem>
```

are predefined drivers for `?feast` reverse communication interface that act on commonly used matrix data storage (dense, banded and compressed sparse row representation), using internal matrix-vector routines and selected inner linear system solvers.

For these interfaces:

- ? indicates the data type of matrix A (and matrix B if any) defined as follows:

s	real, single precision
d	real, double precision
c	complex, single precision
z	complex , double precision

- `<matrix type>` defined as follows:

Value of <i><matrix type></i>	Matrix format	Inner linear system solver used by Extended Eigensolver
sy	(symmetric real)	
he	(Hermitian complex)	Dense LAPACK dense solvers
sb	(symmetric banded real)	
hb	(Hermitian banded complex)	Banded-LAPACK Internal banded solver
scsr	(symmetric real)	
hcsr	(Hermitian complex)	Compressed sparse row PARDISO solver
s	(symmetric real)	
h	(Hermitian complex)	Reverse communications interfaces User defined

- <type of eigenvalue problem>* is:

gv	generalized eigenvalue problem
ev	standard eigenvalue problem

For example, `sfeast_scsrev` is a single-precision routine with a symmetric real matrix stored in sparse compressed-row format for a standard eigenvalue problem, and `zfeast_hrci` is a complex double-precision routine with a Hermitian matrix using the reverse communication interface.

Note that:

- ? can be `s` or `d` if a matrix is real symmetric: *<matrix type>* is `sy`, `sb`, or `scsr`.
- ? can be `c` or `z` if a matrix is complex Hermitian: *<matrix type>* is `he`, `hb`, or `hcsr`.
- ? can be `c` or `z` if the Extended Eigensolver RCI interface is used for solving a complex Hermitian problem.
- ? can be `s` or `d` if the Extended Eigensolver RCI interface is used for solving a real symmetric problem.

feastinit

Initialize Extended Eigensolver input parameters with default values.

Syntax

```
call feastinit (fpm)
```

Include Files

- `mkl.fi`

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

fpm INTEGER

Array, size 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

Extended Eigensolver Input Parameters

The input parameters for Extended Eigensolver routines are contained in an integer array named *fpm*. To call the Extended Eigensolver interfaces, this array should be initialized using the routine *feastinit*.

Param	Default	Description
<i>fpm</i> (1)	0	Specifies whether Extended Eigensolver routines print runtime status.
	<i>fpm</i> (1) = 0	Extended Eigensolver routines do not generate runtime messages at all.
	<i>fpm</i> (1) = 1	Extended Eigensolver routines print runtime status to the screen.
<i>fpm</i> (2)	8	The number of contour points $N_e = 8$ (see the description of FEAST algorithm). Must be one of {3,4,5,6,8,10,12,16,20,24,32,40,48}.
<i>fpm</i> (3)	12	Error trace double precision stopping criteria ε ($\varepsilon = 10^{-fpm(3)}$).
<i>fpm</i> (4)	20	Maximum number of Extended Eigensolver refinement loops allowed. If no convergence is reached within <i>fpm</i> (4) refinement loops, Extended Eigensolver routines return <i>info</i> =2.
<i>fpm</i> (5)	0	User initial subspace. If <i>fpm</i> (5) = 0 then Extended Eigensolver routines generate initial subspace, if <i>fpm</i> (5) = 1 the user supplied initial subspace is used.
<i>fpm</i> (6)	0	Extended Eigensolver stopping test.
	<i>fpm</i> (6) = 0	<p>Extended Eigensolvers are stopped if this residual stopping test is satisfied:</p> <ul style="list-style-type: none"> generalized eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{min} , E_{max}) \ Bx_i\ _1} < \varepsilon$ standard eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{min} , E_{max}) \ x_i\ _1} < \varepsilon$ <p>where <i>mode</i> is the total number of eigenvalues found in the search interval and $\varepsilon = 10^{-fpm(7)}$ for real and complex or $\varepsilon = 10^{-fpm(3)}$ for double precision and double complex.</p>
	<i>fpm</i> (6) = 1	<p>Extended Eigensolvers are stopped if this trace stopping test is satisfied:</p> $\frac{ trace_j - trace_{j-1} }{\max(E_{min} , E_{max})} < \varepsilon$ <p>where <i>trace_j</i> denotes the sum of all eigenvalues found in the search interval [<i>emin</i>, <i>emax</i>] at the <i>j</i>-th Extended Eigensolver iteration:</p>

Param	Default	Description
$trace_j = \sum_{i=1}^M \lambda_i^{(j)}$		
$fpm(7)$	5	Error trace single precision stopping criteria ($10^{-fpm(7)}$).
$fpm(14)$	0	<p>$fpm(14) = 0$ Standard use for Extended Eigensolver routines.</p> <p>$fpm(14) = 1$ Non-standard use for Extended Eigensolver routines: return the computed eigenvectors subspace after one single contour integration.</p>
$fpm(27)$	0	<p>Specifies whether Extended Eigensolver routines check input matrices (applies to CSR format only).</p> <p>$fpm(27) = 0$ Extended Eigensolver routines do not check input matrices.</p> <p>$fpm(27) = 1$ Extended Eigensolver routines check input matrices.</p>
$fpm(28)$	0	Check if matrix B is positive definite. Set $fpm(28) = 1$ to check if B is positive definite.
$fpm(30)$ to $fpm(63)$	-	Reserved for future use.
$fpm(64)$	0	Use the Intel MKL PARDISO solver with the user-defined PARDISO $iparm$ array settings.
<p>NOTE This option can only be used by Extended Eigensolver Predefined Interfaces for Sparse Matrices.</p>		
	$fpm(64) = 0$	Extended Eigensolver routines use the Intel MKL PARDISO default $iparm$ settings defined by calling the <code>pardisoinit</code> subroutine.
	$fpm(64) = 1$	The values from $fpm(65)$ to $fpm(128)$ correspond to $iparm(1)$ to $iparm(64)$ respectively according to the formula $fpm(64 + i) = iparm(i)$ for $i = 1, 2, \dots, 64$.

Extended Eigensolver Output Details

Errors and warnings encountered during a run of the Extended Eigensolver routines are stored in an integer variable, *info*. If the value of the output *info* parameter is not 0, either an error or warning was encountered. The possible return values for the *info* parameter along with the error code descriptions are given in the following table.

Return Codes for info Parameter

<i>info</i>	Classification	Description
202	Error	Problem with size of the system n ($n \leq 0$)
201	Error	Problem with size of initial subspace $m0$ ($m0 \leq 0$ or $m0 > n$)
200	Error	Problem with $emin, emax$ ($emin \geq emax$)

<i>info</i>	Classification	Description
(100+i)	Error	Problem with <i>i</i> -th value of the input Extended Eigensolver parameter (<i>fpm(i)</i>). Only the parameters in use are checked.
4	Warning	Successful return of only the computed subspace after call with <i>fpm(14) = 1</i>
3	Warning	Size of the subspace <i>m0</i> is too small (<i>m0 < m</i>)
2	Warning	No Convergence (number of iteration loops > <i>fpm(4)</i>)
1	Warning	No eigenvalue found in the search interval. See remark below for further details.
0	Successful exit	
-1	Error	Internal error for allocation memory.
-2	Error	Internal error of the inner system solver. Possible reasons: not enough memory for inner linear system solver or inconsistent input.
-3	Error	Internal error of the reduced eigenvalue solver Possible cause: matrix <i>B</i> may not be positive definite. It can be checked by setting <i>fpm(28) = 1</i> before calling an Extended Eigensolver routine, or by using LAPACK routines.
-4	Error	Matrix <i>B</i> is not positive definite.
-(100+i)	Error	Problem with the <i>i</i> -th argument of the Extended Eigensolver interface.

In some extreme cases the return value *info*=1 may indicate that the Extended Eigensolver routine has failed to find the eigenvalues in the search interval. This situation could arise if a very large search interval is used to locate a small and isolated cluster of eigenvalues (i.e. the dimension of the search interval is many orders of magnitude larger than the number of contour points. It is then either recommended to increase the number of contour points *fpm(2)* or simply rescale more appropriately the search interval. Rescaling means the initial problem of finding all eigenvalues the search interval $[\lambda_{\min}, \lambda_{\max}]$ for the standard eigenvalue problem $Ax = \lambda x$ is replaced with the problem of finding all eigenvalues in the search interval $[\lambda_{\min}/t, \lambda_{\max}/t]$ for the standard eigenvalue problem $(A/t)x = (\lambda/t)x$ where *t* is a scaling factor.

Extended Eigensolver RCI Routines

If you do not require specific linear system solvers or matrix storage schemes, you can skip this section and go directly to [Extended Eigensolver Predefined Interfaces](#).

Extended Eigensolver RCI Interface Description

The Extended Eigensolver RCI interfaces can be used to solve standard or generalized eigenvalue problems, and are independent of the format of the matrices. As mentioned earlier, the Extended Eigensolver algorithm is based on the contour integration techniques of the matrix resolvent $G(\sigma) = (\sigma B - A)^{-1}$ over a circle. For solving a generalized eigenvalue problem, Extended Eigensolver has to perform one or more of the following operations at each contour point denoted below by Z_e :

- Factorize the matrix $(Z_e * B - A)$
- Solve the linear system $(Z_e * B - A)X = Y$ or $(Z_e * B - A)^H X = Y$ with multiple right hand sides, where H means transpose conjugate
- Matrix-matrix multiply $BX = Y$ or $AX = Y$

For solving a standard eigenvalue problem, replace the matrix *B* with the identity matrix *I*.

The primary aim of RCI interfaces is to isolate these operations: the linear system solver, factorization of the matrix resolvent at each contour point, and matrix-matrix multiplication. This gives universality to RCI interfaces as they are independent of data structures and the specific implementation of the operations like matrix-vector multiplication or inner system solvers. However, this approach requires some additional effort when calling the interface. In particular, operations listed above are performed by routines that you supply on data structures that you find most appropriate for the problem at hand.

To initialize an Extended Eigensolver RCI routine, set the job indicator (*ijob*) parameter to the value -1. When the routine requires the results of an operation, it generates a special value of *ijob* to indicate the operation that needs to be performed. The routine also returns *ze*, the coordinate along the complex contour, the values of array *work* or *workc*, and the number of columns to be used. Your subroutine then must perform the operation at the given contour point *ze*, store the results in prescribed array, and return control to the Extended Eigensolver RCI routine.

The following pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a real symmetric problem:

```
Ijob=-1 ! initialization
do while (ijob/=0)
  call ?feast_srcf(ijob, N, Ze, work1, work2, Aq, Bq,
    &fpm, epsout, loop, Emin, Emax, M0, E, lambda, q, res, info)

  select case(ijob)
  case(10) !! Factorize the complex matrix (ZeB-A)
  . . . . . <<< user entry

  case(11) !! Solve the complex linear system (ZeB-A)x=work2(1:N,1:M0) result in work2
  . . . . . <<< user entry

  case(30) !! Perform multiplication A*q(1:N,i:j) result in work1(1:N,i:j)
    !! where i=fpm(24) and j=fpm(24)+fpm(25)-1
  . . . . . <<< user entry
  case(40) !! Perform multiplication B*q(1:N,i:j) result in work1(1:N,i:j)
    !! where i=fpm(24) and j=fpm(24)+fpm(25)-1
  . . . . . <<< user entry

  end select
end do
```

NOTE

The ? option in ?feast in the pseudocode given above should be replaced by either *s* or *d*, depending on the matrix data type of the eigenvalue system.

The next pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a complex Hermitian problem:

```
Ijob=-1 ! initialization
do while (ijob/=0)
  call ?feast_hrcf(ijob, N, Ze, work1, work2, Aq, Bq,
    &fpm, epsout, loop, Emin, Emax, M0, E, lambda, q, res, info)

  select case(ijob)
  case(10) !! Factorize the complex matrix (ZeB-A)
  . . . . . <<< user entry
  case (11)!! Solve the linear system (ZeB-A)y=work2 (1:N, 1:M0) result in work2
  . . . . . <<< user entry
  case (20)!! Factorize ( if needed by case (21)) the complex matrix (ZeB-A)^H
  !!ATTENTION: This option requires additional memory storage
  !! (i.e . the resulting matrix from case (10) cannot be overwritten)
  . . . . . <<< user entry
  case (21) !! Solve the linear system (ZeB-A)^Hy=work2(1:N, 1:M0) result in work2
```

```

!!REMARK: case (20) becomes obsolete if this solve can be performed
!! using the factorization in case (10)
. . . . . <<< user entry
    case(30) !! Perform multiplication A*q(1:N,i:j) result in work1(1:N,i:j)
              !! where i=fpm(24) and j=fpm(25)+fpm(24)-1
. . . . . <<< user entry
    case(40) !! Perform multiplication B*q(1:N,i:j) result in work1(1:N,i:j)
              !! where i=fpm(24) and j=fpm(25)+fpm(24)-1
. . . . . <<< user entry

end select
end do

```

NOTE

The `?feast` option in the pseudocode given above should be replaced by either `c` or `z`, depending on the matrix data type of the eigenvalue system.

If `case(20)` can be avoided, performance could be up to twice as fast, and Extended Eigensolver functionality would use half of the memory.

If an iterative solver is used along with a preconditioner, the factorization of the preconditioner could be performed with `ijob = 10` (and `ijob = 20` if applicable) for a given value of Z_e , and the associated iterative solve would then be performed with `ijob = 11` (and `ijob = 21` if applicable).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`?feast_src`/`?feast_hrci`

Extended Eigensolver RCI interface.

Syntax

```
call sfeast_src (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)
```

```
call dfeast_src (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)
```

```
call cfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)
```

```
call zfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,
lambda, q, m, res, info)
```

Include Files

- `mkl.fi`

Description

Compute eigenvalues as described in [Extended Eigensolver RCI Interface Description](#).

Input Parameters

<i>ijob</i>	INTEGER	Job indicator variable. On entry, a call to ?feast_src1/?feast_hrc1 with <i>ijob</i> =-1 initializes the eigensolver.
<i>n</i>	INTEGER	Sets the size of the problem. $n > 0$.
<i>work</i>	REAL for sfeast_src1 DOUBLE PRECISION for dfeast_src1 COMPLEX for cfeast_hrc1 COMPLEX*16 for zfeast_hrc1	Workspace array of size n by $m0$.
<i>workc</i>	COMPLEX for sfeast_src1 and cfeast_hrc1 COMPLEX*16 for dfeast_src1 and zfeast_hrc1	Workspace array of size n by $m0$.
<i>aq, sq</i>	REAL for sfeast_src1 DOUBLE PRECISION for dfeast_src1 COMPLEX for cfeast_hrc1 COMPLEX*16 for zfeast_hrc1	Workspace arrays of size $m0$ by $m0$.
<i>fpm</i>	INTEGER	Array, size of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin, emax</i>	REAL for sfeast_src1 and cfeast_hrc1 DOUBLE PRECISION for dfeast_src1 and zfeast_hrc1	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.
<i>m0</i>	INTEGER	On entry, specifies the initial guess for subspace size to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i> =3.
<i>q</i>	REAL for sfeast_src1 DOUBLE PRECISION for dfeast_src1 COMPLEX for cfeast_hrc1 COMPLEX*16 for zfeast_hrc1	

On entry, if $fpm(5) = 1$, the array q of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

ijob

On exit, the parameter carries the status flag that indicates the condition of the return. The status information is divided into three categories:

1. A zero value indicates successful completion of the task.
2. A positive value indicates that the solver requires a matrix-vector multiplication or solving a specific system with a complex coefficient.
3. A negative value indicates successful initiation.

A non-zero value of *ijob* specifically means the following:

- *ijob* = 10 - factorize the complex matrix $Z_e * B - A$ at a given contour point Z_e and return the control to the `?feast_src`/`?feast_hrci` routine where Z_e is a complex number meaning contour point and its value is defined internally in `?feast_src`/`?feast_hrci`.
- *ijob* = 11 - solve the complex linear system $(Z_e * B - A) * y = workc(n, m0)$, put the solution in `workc(n, m0)` and return the control to the `?feast_src`/`?feast_hrci` routine.
- *ijob* = 20 - factorize the complex matrix $(Z_e * B - A)^H$ at a given contour point Z_e and return the control to the `?feast_src`/`?feast_hrci` routine where Z_e is a complex number meaning contour point and its value is defined internally in `?feast_src`/`?feast_hrci`.

The symbol X^H means transpose conjugate of matrix X .

- *ijob* = 21 - solve the complex linear system $(Z_e * B - A)^H * y = workc(n, m0)$, put the solution in `workc(n, m0)` and return the control to the `?feast_src`/`?feast_hrci` routine. The case *ijob*=20 becomes obsolete if the solve can be performed using the factorization computed for *ijob*=10.

The symbol X^H mean transpose conjugate of matrix X .

- *ijob* = 30 - multiply matrix A by `q(n, i:j)`, put the result in `work(n, i:j)`, and return the control to the `?feast_src`/`?feast_hrci` routine.
i is `fpm(25)`, and *j* is `fpm(24) + fpm(25) - 1`.
- *ijob* = 40 - multiply matrix B by `q(n, i:j)`, put the result in `work(n, i:j)` and return the control to the `?feast_src`/`?feast_hrci` routine. If a standard eigenvalue problem is solved, just return `work = q`.
i is `fpm(25)`, and *j* is `fpm(24) + fpm(25) - 1`.
- *ijob* = -2 - rerun the `?feast_src`/`?feast_hrci` task with the same parameters.

ze

COMPLEX for `sfeast_src` and `cfeast_hrci`

COMPLEX*16 for `dfeast_src` and `zfeast_hrci`

Defines the coordinate along the complex contour. All values of *ze* are generated by `?feast_src`/`?feast_hrci` internally.

fpm

On output, contains coordinates of columns of work array needed for iterative refinement. (See [Extended Eigensolver RCI Interface Description](#).)

<i>epsout</i>	<p>REAL for <i>sfeast_src</i> and <i>cfeast_hrci</i></p> <p>DOUBLE PRECISION for <i>dfeast_src</i> and <i>zfeast_hrci</i></p> <p>On output, contains the relative error on the trace: $trace_i - trace_{i-1} / \max(emin , emax)$</p>
<i>loop</i>	<p>INTEGER</p> <p>On output, contains the number of refinement loop executed. Ignored on input.</p>
<i>lambda</i>	<p>REAL for <i>sfeast_src</i> and <i>cfeast_hrci</i></p> <p>DOUBLE PRECISION for <i>dfeast_src</i> and <i>zfeast_hrci</i></p> <p>Array of length <i>m0</i>. On output, the first <i>m</i> entries of <i>lambda</i> are eigenvalues found in the interval.</p>
<i>q</i>	<p>On output, <i>q</i> contains all eigenvectors corresponding to <i>lambda</i>.</p>
<i>m</i>	<p>INTEGER</p> <p>The total number of eigenvalues found in the interval [<i>emin</i>, <i>emax</i>]: $0 \leq m \leq m0$.</p>
<i>res</i>	<p>REAL for <i>sfeast_src</i> and <i>cfeast_hrci</i></p> <p>DOUBLE PRECISION for <i>dfeast_src</i> and <i>zfeast_hrci</i></p> <p>Array of length <i>m0</i>. On exit, the first <i>m</i> components contain the relative residual vector:</p> <ul style="list-style-type: none"> generalized eigenvalue problem: $\frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1}$ standard eigenvalue problem: $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1}$ <p>for $i=1, 2, \dots, m$, and where <i>m</i> is the total number of eigenvalues found in the search interval.</p>
<i>info</i>	<p>INTEGER</p> <p>If <i>info</i>=0, the execution is successful. If <i>info</i>≠0, see Output Eigensolver info Details.</p>

Extended Eigensolver Predefined Interfaces

The predefined interfaces include routines for standard and generalized eigenvalue problems, and for dense, banded, and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Dense	?feast_syev ?feast_heev	?feast_sygv ?feast_hegv
Banded	?feast_sbev ?feast_hbev	?feast_sbgv ?feast_hbgv
Sparse	?feast_scsrev ?feast_hcsrev	?feast_scsrgv ?feast_hcsrgv

Matrix Storage

The symmetric and Hermitian matrices used in Extended Eigensolvers predefined interfaces can be stored in full, band, and sparse formats.

- In the full storage format (described in [Full Storage](#) in additional detail) you store all elements, all of the elements in the upper triangle of the matrix, or all of the elements in the lower triangle of the matrix.
- In the band storage format (described in [Band storage](#) in additional detail), you store only the elements along a diagonal band of the matrix.
- In the sparse format (described in [Storage Arrays for a Matrix in CSR Format \(3-Array Variation\)](#)), you store only the non-zero elements of the matrix.

In generalized eigenvalue systems you must use the same family of storage format for both matrices A and B . The bandwidth can be different for the banded format (klb can be different from kla), and the position of the non-zero elements can also be different for the sparse format (CSR coordinates ib and jb can be different from ia and ja).

?feast_syev/?feast_heev

Extended Eigensolver interface for standard eigenvalue problem with dense matrices.

Syntax

```
call sfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call dfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call cfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call zfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	<p>CHARACTER*1</p> <p>Must be 'U' or 'L' or 'F' .</p> <p>If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i>.</p> <p>If <i>uplo</i> = 'F', <i>a</i> stores the full matrix <i>A</i>.</p>
<i>n</i>	<p>INTEGER</p> <p>Sets the size of the problem. $n > 0$.</p>
<i>a</i>	<p>REAL for <i>sfeast_syev</i></p> <p>DOUBLE PRECISION for <i>dfeast_syev</i></p> <p>COMPLEX for <i>cfeast_heev</i></p> <p>COMPLEX*16 for <i>zfeast_heev</i></p> <p>Array of dimension <i>lda</i> by <i>n</i>, contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i>, as specified by <i>uplo</i></p>
<i>lda</i>	<p>INTEGER</p> <p>The leading dimension of the array <i>a</i>. Must be at least $\max(1, n)$.</p>
<i>fpm</i>	<p>INTEGER</p> <p>Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.</p>
<i>emin, emax</i>	<p>REAL for <i>sfeast_syev</i> and <i>cfeast_heev</i></p> <p>DOUBLE PRECISION for <i>dfeast_syev</i> and <i>zfeast_heev</i></p> <p>The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.</p>
<i>m0</i>	<p>INTEGER</p> <p>On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where <i>m</i> is the total number of eigenvalues located in the interval [<i>emin</i>, <i>emax</i>]. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i>=3.</p>
<i>x</i>	<p>REAL for <i>sfeast_syev</i></p> <p>DOUBLE PRECISION for <i>dfeast_syev</i></p> <p>COMPLEX for <i>cfeast_heev</i></p> <p>COMPLEX*16 for <i>zfeast_heev</i></p> <p>On entry, if <i>fpm</i>(5)=1, the array <i>x</i>(<i>n</i>, <i>m</i>) contains a basis of guess subspace where <i>n</i> is the order of the input matrix.</p>

Output Parameters

<i>epsout</i>	<p>REAL for <code>sfeast_syev</code> and <code>cfeast_heev</code></p> <p>DOUBLE PRECISION for <code>dfeast_syev</code> and <code>zfeast_heev</code></p> <p>On output, contains the relative error on the trace: $trace_i - trace_{i-1} / \max(e_{min} , e_{max})$</p>
<i>loop</i>	<p>INTEGER</p> <p>On output, contains the number of refinement loop executed. Ignored on input.</p>
<i>e</i>	<p>REAL for <code>sfeast_syev</code> and <code>cfeast_heev</code></p> <p>DOUBLE PRECISION for <code>dfeast_syev</code> and <code>zfeast_heev</code></p> <p>Array of length <i>m0</i>. On output, the first <i>m</i> entries of <i>e</i> are eigenvalues found in the interval.</p>
<i>x</i>	<p>On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i>, with the <i>i</i>-th column of <i>x</i> holding the eigenvector associated with <i>e</i>(<i>i</i>).</p>
<i>m</i>	<p>INTEGER</p> <p>The total number of eigenvalues found in the interval [<i>e_{min}</i>, <i>e_{max}</i>]: $0 \leq m \leq m0$.</p>
<i>res</i>	<p>REAL for <code>sfeast_syev</code> and <code>cfeast_heev</code></p> <p>DOUBLE PRECISION for <code>dfeast_syev</code> and <code>zfeast_heev</code></p> <p>Array of length <i>m0</i>. On exit, the first <i>m</i> components contain the relative residual vector:</p> $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{min} , E_{max}) \ x_i\ _1}$ <p>for <i>i</i>=1, 2, ..., <i>m</i>, and where <i>m</i> is the total number of eigenvalues found in the search interval.</p>
<i>info</i>	<p>INTEGER</p> <p>If <i>info</i>=0, the execution is successful. If <i>info</i>≠ 0, see Output Eigensolver info Details.</p>

?feast_sygv/?feast_hegv

Extended Eigensolver interface for generalized eigenvalue problem with dense matrices.

Syntax

call `sfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)`

call `dfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)`

```
call cfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

```
call zfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

Input Parameters

<i>uplo</i>	CHARACTER*1 Must be 'U' or 'L' or 'F' . If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.
<i>n</i>	INTEGER Sets the size of the problem. $n > 0$.
<i>a</i>	REAL for sfeast_sygv DOUBLE PRECISION for dfeast_sygv COMPLEX for cfeast_hegv COMPLEX*16 for zfeast_hegv Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	INTEGER The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>b</i>	REAL for sfeast_sygv DOUBLE PRECISION for dfeast_sygv COMPLEX for cfeast_hegv COMPLEX*16 for zfeast_hegv Array of dimension <i>ldb</i> by <i>n</i> , contains either full matrix <i>B</i> or upper or lower triangular part of the matrix <i>B</i> , as specified by <i>uplo</i>
<i>ldb</i>	INTEGER The leading dimension of the array <i>B</i> . Must be at least $\max(1, n)$.
<i>fpm</i>	INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

emin, emax

REAL for `sfeast_sygv` and `cfeast_hegv`

DOUBLE PRECISION for `dfeast_sygv` and `zfeast_hegv`

The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return `info=3`.

x

REAL for `sfeast_sygv`

DOUBLE PRECISION for `dfeast_sygv`

COMPLEX for `cfeast_hegv`

COMPLEX*16 for `zfeast_hegv`

On entry, if `fpm(5)=1`, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout

REAL for `sfeast_sygv` and `cfeast_hegv`

DOUBLE PRECISION for `dfeast_sygv` and `zfeast_hegv`

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for `sfeast_sygv` and `cfeast_hegv`

DOUBLE PRECISION for `dfeast_sygv` and `zfeast_hegv`

Array of length $m0$. On output, the first m entries of e are eigenvalues found in the interval.

x

On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e , with the i -th column of x holding the eigenvector associated with $e(i)$.

m

INTEGER

The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.

res

REAL for `sfeast_sygv` and `cfeast_hegv`

DOUBLE PRECISION for `dfeast_sygv` and `zfeast_hegv`

Array of length $m0$. On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

INTEGER

If *info*=0, the execution is successful. If *info*≠ 0, see [Output Eigensolver info Details](#).

?feast_sbev/?feast_hbev

Extended Eigensolver interface for standard eigenvalue problem with banded matrices.

Syntax

call sfeast_sbev(*uplo*, *n*, *kla*, *a*, *lda*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call dfeast_sbev(*uplo*, *n*, *kla*, *a*, *lda*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call cfeast_hbev(*uplo*, *n*, *kla*, *a*, *lda*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call zfeast_hbev(*uplo*, *n*, *kla*, *a*, *lda*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo

CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

INTEGER

Sets the size of the problem. $n > 0$.

kla

INTEGER

The number of super- or sub-diagonals within the band in *A* ($kla \geq 0$).

<i>a</i>	<p>REAL for <code>sfeast_sbev</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbev</code></p> <p>COMPLEX for <code>cfeast_hbev</code></p> <p>COMPLEX*16 for <code>zfeast_hbev</code></p> <p>Array of dimension <i>lda</i> by <i>n</i>, contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i>, as specified by <i>uplo</i></p>
<i>lda</i>	<p>INTEGER</p> <p>The leading dimension of the array <i>a</i>. Must be at least $\max(1, n)$.</p>
<i>fpm</i>	<p>INTEGER</p> <p>Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.</p>
<i>emin, emax</i>	<p>REAL for <code>sfeast_sbev</code> and <code>cfeast_hbev</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbev</code> and <code>zfeast_hbev</code></p> <p>The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.</p>
<i>m0</i>	<p>INTEGER</p> <p>On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where <i>m</i> is the total number of eigenvalues located in the interval [<i>emin</i>, <i>emax</i>]. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i>=3.</p>
<i>x</i>	<p>REAL for <code>sfeast_sbev</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbev</code></p> <p>COMPLEX for <code>cfeast_hbev</code></p> <p>COMPLEX*16 for <code>zfeast_hbev</code></p> <p>On entry, if <i>fpm</i>(5)=1, the array <i>x</i>(<i>n</i>, <i>m</i>) contains a basis of guess subspace where <i>n</i> is the order of the input matrix.</p>

Output Parameters

<i>epsout</i>	<p>REAL for <code>sfeast_sbev</code> and <code>cfeast_hbev</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbev</code> and <code>zfeast_hbev</code></p> <p>On output, contains the relative error on the trace: $trace_i - trace_{i-1} / \max(emin , emax)$</p>
<i>loop</i>	<p>INTEGER</p> <p>On output, contains the number of refinement loop executed. Ignored on input.</p>
<i>e</i>	<p>REAL for <code>sfeast_sbev</code> and <code>cfeast_hbev</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbev</code> and <code>zfeast_hbev</code></p>

	Array of length $m0$. On output, the first m entries of e are eigenvalues found in the interval.
x	On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e , with the i -th column of x holding the eigenvector associated with $e(i)$.
m	INTEGER The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.
res	REAL for <code>sfeast_sbev</code> and <code>cfeast_hbev</code> DOUBLE PRECISION for <code>dfeast_sbev</code> and <code>zfeast_hbev</code> Array of length $m0$. On exit, the first m components contain the relative residual vector: $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1}$ for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.
$info$	INTEGER If $info=0$, the execution is successful. If $info \neq 0$, see Output Eigensolver info Details .

?feast_sbgv/?feast_hbgv

Extended Eigensolver interface for generalized eigenvalue problem with banded matrices.

Syntax

```
call sfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call dfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call cfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

```
call zfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m, res, info)
```

Include Files

- `mk1.fi`

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices *A* and *B* must use the same family of storage format. The bandwidth, however, can be different (*klb* can be different from *kla*).

Input Parameters

<i>uplo</i>	<p>CHARACTER*1</p> <p>Must be 'U' or 'L' or 'F' .</p> <p>If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively.</p> <p>If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively.</p> <p>If <i>UPLO</i>= 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.</p>
<i>n</i>	<p>INTEGER</p> <p>Sets the size of the problem. $n > 0$.</p>
<i>kla</i>	<p>INTEGER</p> <p>The number of super- or sub-diagonals within the band in <i>A</i> ($kla \geq 0$).</p>
<i>a</i>	<p>REAL for sfeast_sbgv</p> <p>DOUBLE PRECISION for dfeast_sbgv</p> <p>COMPLEX for cfeast_hbgv</p> <p>COMPLEX*16 for zfeast_hbgv</p> <p>Array of dimension <i>lda</i> by <i>n</i>, contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i>, as specified by <i>uplo</i></p>
<i>lda</i>	<p>INTEGER</p> <p>The leading dimension of the array <i>a</i>. Must be at least $\max(1, n)$.</p>
<i>klb</i>	<p>INTEGER</p> <p>The number of super- or sub-diagonals within the band in <i>B</i> ($klb \geq 0$).</p>
<i>b</i>	<p>REAL for sfeast_sbgv</p> <p>DOUBLE PRECISION for dfeast_sbgv</p> <p>COMPLEX for cfeast_hbgv</p> <p>COMPLEX*16 for zfeast_hbgv</p> <p>Array of dimension <i>ldb</i> by <i>n</i>, contains either full matrix <i>B</i> or upper or lower triangular part of the matrix <i>B</i>, as specified by <i>uplo</i></p>
<i>ldb</i>	<p>INTEGER</p> <p>The leading dimension of the array <i>B</i>. Must be at least $\max(1, n)$.</p>
<i>fpm</i>	<p>INTEGER</p>

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

<i>emin, emax</i>	<p>REAL for <code>sfeast_sbgv</code> and <code>cfeast_hbgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbgv</code> and <code>zfeast_hbgv</code></p> <p>The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.</p>
<i>m0</i>	<p>INTEGER</p> <p>On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <code>info=3</code>.</p>
<i>x</i>	<p>REAL for <code>sfeast_sbgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbgv</code></p> <p>COMPLEX for <code>cfeast_hbgv</code></p> <p>COMPLEX*16 for <code>zfeast_hbgv</code></p> <p>On entry, if <code>fpm(5)=1</code>, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.</p>

Output Parameters

<i>epsout</i>	<p>REAL for <code>sfeast_sbgv</code> and <code>cfeast_hbgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbgv</code> and <code>zfeast_hbgv</code></p> <p>On output, contains the relative error on the trace: $trace_i - trace_{i-1} / \max(emin , emax)$</p>
<i>loop</i>	<p>INTEGER</p> <p>On output, contains the number of refinement loop executed. Ignored on input.</p>
<i>e</i>	<p>REAL for <code>sfeast_sbgv</code> and <code>cfeast_hbgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbgv</code> and <code>zfeast_hbgv</code></p> <p>Array of length $m0$. On output, the first m entries of e are eigenvalues found in the interval.</p>
<i>x</i>	<p>On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e, with the i-th column of x holding the eigenvector associated with $e(i)$.</p>
<i>m</i>	<p>INTEGER</p> <p>The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.</p>
<i>res</i>	<p>REAL for <code>sfeast_sbgv</code> and <code>cfeast_hbgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_sbgv</code> and <code>zfeast_hbgv</code></p>

Array of length $m0$. On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

INTEGER

If *info*=0, the execution is successful. If *info*≠ 0, see [Output Eigensolver info Details](#).

[?feast_scsrev/?feast_hcsrev](#)

Extended Eigensolver interface for standard eigenvalue problem with sparse matrices.

Syntax

call sfeast_scsrev(*uplo*, *n*, *a*, *ia*, *ja*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call dfeast_scsrev(*uplo*, *n*, *a*, *ia*, *ja*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call cfeast_hcsrev(*uplo*, *n*, *a*, *ia*, *ja*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

call zfeast_hcsrev(*uplo*, *n*, *a*, *ia*, *ja*, *fpm*, *epsout*, *loop*, *emin*, *emax*, *m0*, *e*, *x*, *m*, *res*, *info*)

Include Files

- mkl.fi

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo

CHARACTER*1

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

INTEGER

Sets the size of the problem. $n > 0$.

a

REAL for sfeast_scsrev

DOUBLE PRECISION for dfeast_scsrev

	COMPLEX for <code>cfeast_hcsrev</code> COMPLEX*16 for <code>zfeast_hcsrev</code>
	Array containing the nonzero elements of either the full matrix A or the upper or lower triangular part of the matrix A , as specified by <code>uplo</code> .
<code>ia</code>	INTEGER Array of length $n + 1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(n + 1)$ is equal to the number of non-zeros plus one.
<code>ja</code>	INTEGER Array containing the column indices for each non-zero element of the matrix A being represented in the array a . Its length is equal to the length of the array a .
<code>fpm</code>	INTEGER Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<code>emin, emax</code>	REAL for <code>sfeast_scsrev</code> and <code>cfeast_hcsrev</code> DOUBLE PRECISION for <code>dfeast_scsrev</code> and <code>zfeast_hcsrev</code> The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.
<code>m0</code>	INTEGER On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <code>info=3</code> .
<code>x</code>	REAL for <code>sfeast_scsrev</code> DOUBLE PRECISION for <code>dfeast_scsrev</code> COMPLEX for <code>cfeast_hcsrev</code> COMPLEX*16 for <code>zfeast_hcsrev</code> On entry, if <code>fpm(5) = 1</code> , the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

<code>fpm</code>	On output, the last 64 values correspond to Intel MKL PARDISO <code>iparm(1)</code> to <code>iparm(64)</code> (regardless of the value of <code>fpm(64)</code> on input).
<code>epsout</code>	REAL for <code>sfeast_scsrev</code> and <code>cfeast_hcsrev</code> DOUBLE PRECISION for <code>dfeast_scsrev</code> and <code>zfeast_hcsrev</code>

	On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(e_{min} , e_{max})$
<i>loop</i>	INTEGER On output, contains the number of refinement loop executed. Ignored on input.
<i>e</i>	REAL for <code>sfeast_scsrev</code> and <code>cfeast_hcsrev</code> DOUBLE PRECISION for <code>dfeast_scsrev</code> and <code>zfeast_hcsrev</code> Array of length <i>m0</i> . On output, the first <i>m</i> entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the <i>i</i> -th column of <i>x</i> holding the eigenvector associated with <i>e</i> (<i>i</i>).
<i>m</i>	INTEGER The total number of eigenvalues found in the interval [<i>e_{min}</i> , <i>e_{max}</i>]: $0 \leq m \leq m0$.
<i>res</i>	REAL for <code>sfeast_scsrev</code> and <code>cfeast_hcsrev</code> DOUBLE PRECISION for <code>dfeast_scsrev</code> and <code>zfeast_hcsrev</code> Array of length <i>m0</i> . On exit, the first <i>m</i> components contain the relative residual vector: $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{min} , E_{max}) \ x_i\ _1}$ for <i>i</i> =1, 2, ..., <i>m</i> , and where <i>m</i> is the total number of eigenvalues found in the search interval.
<i>info</i>	INTEGER If <i>info</i> =0, the execution is successful. If <i>info</i> ≠ 0, see Output Eigensolver info Details .

?feast_scsrgv/?feast_hcsrgv

Extended Eigensolver interface for generalized eigenvalue problem with sparse matrices.

Syntax

```
call sfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
call dfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
call cfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
call zfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
```

Include Files

- `mkl.fi`

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices A and B must use the same family of storage format. The position of the non-zero elements can be different (CSR coordinates ib and jb can be different from ia and ja).

Input Parameters

<code>uplo</code>	CHARACTER*1 Must be 'U' or 'L' or 'F' . If <code>UPLO = 'U'</code> , a and b store the upper triangular parts of A and B respectively. If <code>UPLO = 'L'</code> , a and b store the lower triangular parts of A and B respectively. If <code>UPLO = 'F'</code> , a and b store the full matrices A and B respectively.
<code>n</code>	INTEGER Sets the size of the problem. $n > 0$.
<code>a</code>	REAL for <code>sfeast_scsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> COMPLEX for <code>cfeast_hcsrgv</code> COMPLEX*16 for <code>zfeast_hcsrgv</code> Array containing the nonzero elements of either the full matrix A or the upper or lower triangular part of the matrix A , as specified by <code>uplo</code> .
<code>ia</code>	INTEGER Array of length $n + 1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(n + 1)$ is equal to the number of non-zeros plus one.
<code>ja</code>	INTEGER Array containing the column indices for each non-zero element of the matrix A being represented in the array a . Its length is equal to the length of the array a .
<code>b</code>	REAL for <code>sfeast_scsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> COMPLEX for <code>cfeast_hcsrgv</code> COMPLEX*16 for <code>zfeast_hcsrgv</code>

	Array of dimension ldb by $*$, contains either full matrix B or upper or lower triangular part of the matrix B , as specified by $uplo$
ib	INTEGER Array of length $n + 1$, containing indices of elements in the array b , such that $ib(i)$ is the index in the array b of the first non-zero element from the row i . The value of the last element $ib(n + 1)$ is equal to the number of non-zeros plus one.
jb	INTEGER Array containing the column indices for each non-zero element of the matrix B being represented in the array b . Its length is equal to the length of the array b .
fpm	INTEGER Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
$emin, emax$	REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code> The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.
$m0$	INTEGER On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <code>info=3</code> .
x	REAL for <code>sfeast_scsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> COMPLEX for <code>cfeast_hcsrgv</code> COMPLEX*16 for <code>zfeast_hcsrgv</code> On entry, if $fpm(5) = 1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

fpm	On output, the last 64 values correspond to Intel MKL PARDISO $iparm(1)$ to $iparm(64)$ (regardless of the value of $fpm(64)$ on input).
$epsout$	REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code> On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(emin , emax)$
$loop$	INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

<i>e</i>	<p>REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code></p> <p>Array of length $m0$. On output, the first m entries of e are eigenvalues found in the interval.</p>
<i>x</i>	<p>On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e, with the i-th column of x holding the eigenvector associated with $e(i)$.</p>
<i>m</i>	<p>INTEGER</p> <p>The total number of eigenvalues found in the interval $[e_{min}, e_{max}]$: $0 \leq m \leq m0$.</p>
<i>res</i>	<p>REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code></p> <p>DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code></p> <p>Array of length $m0$. On exit, the first m components contain the relative residual vector:</p> $\frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1}$ <p>for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.</p>
<i>info</i>	<p>INTEGER</p> <p>If $info=0$, the execution is successful. If $info \neq 0$, see Output Eigensolver info Details.</p>

Vector Mathematical Functions

This chapter describes Intel® MKL Vector Mathematics functions (VM), which compute a mathematical function of each of the vector elements. VM includes a set of highly optimized functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- **VM Mathematical Functions** compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- **VM Pack/Unpack Functions** convert to and from vectors with positive increment indexing, vector indexing, and mask indexing (see [Appendix B](#) for details on vector indexing methods).
- **VM Service Functions** set/get the accuracy modes and the error codes, and free memory.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

The Intel MKL interfaces are given in `mkl_vml.f90`; the `mkl_vml.fi` include file available in the previous versions of Intel MKL is retained for backward compatibility

Examples that demonstrate how to use the VM functions are located in:

```
{MKL}/examples/vmlf/source
```

See VM performance and accuracy data in the online VM Performance and Accuracy Data document available at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Data Types, Accuracy Modes, and Performance Tips

VM includes mathematical and pack/unpack vector functions for single and double precision vector arguments of real and complex types. Intel MKL provides Fortran and C interfaces for all VM functions, including the associated service functions. The Function Naming Conventions section below shows how to call these functions.

Performance depends on a number of factors, including vectorization and threading overhead. The recommended usage is as follows:

- Use VM for vector lengths larger than 40 elements.
- Use the Intel® Compiler for vector lengths less than 40 elements.

All VM vector functions support the following accuracy modes:

- High Accuracy (HA), the default mode
- Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits
- Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

Note that using the EP mode does not guarantee accurate processing of corner cases and special values. Although the default accuracy is HA, LA is sufficient in most cases. For applications that require less accuracy (for example, media applications, some Monte Carlo simulations, etc.), the EP mode may be sufficient.

VM handles special values in accordance with the C99 standard [C99].

Intel MKL offers both functions and environment variables to switch between modes for VM. See the *Intel MKL User's Guide* for details about the environment variables. Use the `vmlsetmode(mode)` function (see [Table "Values of the mode Parameter"](#)) to switch between the HA, LA, and EP modes. The `vmlgetmode()` function returns the current mode.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[Function Naming Conventions](#)

Function Naming Conventions

The VM function names are lowercase.

The VM mathematical and pack/unpack function names have the following structure:

`v[m]<?><name><mod>`

where

- `v` is a prefix indicating vector operations.
- `[m]` is an optional prefix for mathematical functions that indicates additional argument to specify a VM mode for a given function call (see `vmlsetmode` for possible values and their description).
- `<?>` is a precision prefix that indicates one of the following the data types:

<code>s</code>	REAL (KIND=4).
<code>d</code>	REAL (KIND=8).
<code>c</code>	COMPLEX (KIND=4).
<code>z</code>	COMPLEX (KIND=8).

- `<name>` indicates the function short name. See examples in [Table "VM Mathematical Functions"](#).
- `<mod>` field is present only in the pack/unpack functions and indicates the indexing method used:

<code>i</code>	indexing with a positive increment
<code>v</code>	indexing with an index vector

m indexing with a mask vector.

The VM service function names have the following structure:

vm<name>

where

<name> indicates the function short name. See examples in [Table "VM Service Functions"](#).

To call VM functions from an application program, use conventional function calls. For example, call the vector single precision real exponential function as

```
call vsexp ( n, a, y )
```

```
call vmsexp ( n, a, y, mode ) with a specified mode
```

Function Interfaces

VM interfaces include the function names and argument lists. The following sections describe the interfaces for the VM functions. Note that some of the functions have multiple input and output arguments

Some VM functions may also take scalar arguments as input. See the function description for the naming conventions of such arguments.

VM Mathematical Functions

```
call v<?><name>( n, a, [scalar input arguments],y )
call v<?><name>( n, a, b, [scalar input arguments],y )
call v<?><name>( n, a, y, z )
call vm<?><name>( n, a, [scalar input arguments],y, mode )
call vm<?><name>( n, a, b, [scalar input arguments],y, mode )
call vm<?><name>( n, a, y, z, mode )
```

Pack Functions

```
call v<?>packi( n, a, inca, y )
call v<?>packv( n, a, ia, y )
call v<?>packm( n, a, ma, y )
```

Unpack Functions

```
call v<?>unpacki( n, a, y, incy )
call v<?>unpackv( n, a, y, iy )
call v<?>unpackm( n, a, y, my )
```

Service Functions

```
oldmode = vmlsetmode( mode )
mode = vmlgetmode( )
olderr = vmlseterrstatus ( err )
err = vmlgeterrstatus( )
olderr = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

Note that *oldmode*, *oldcerr*, and *oldcallback* refer to settings prior to the call.

Input Parameters

<i>n</i>	number of elements to be calculated
<i>a</i>	first input vector
<i>b</i>	second input vector
<i>inca</i>	vector increment for the input vector <i>a</i>
<i>ia</i>	index vector for the input vector <i>a</i>
<i>ma</i>	mask vector for the input vector <i>a</i>
<i>incy</i>	vector increment for the output vector <i>y</i>
<i>iy</i>	index vector for the output vector <i>y</i>
<i>my</i>	mask vector for the output vector <i>y</i>
<i>err</i>	error code
<i>mode</i>	VM mode
<i>callback</i>	address of the callback function

Output Parameters

<i>y</i>	first output vector
<i>z</i>	second output vector
<i>err</i>	error code
<i>mode</i>	VM mode
<i>olderr</i>	former error code
<i>oldmode</i>	former VM mode
<i>callback</i>	address of the callback function
<i>oldcallback</i>	address of the former callback function

See the data types of the parameters used in each function in the respective function description section. All the Intel MKL VM mathematical functions can perform in-place operations.

Vector Indexing Methods

VM mathematical functions work only with unit stride. To accommodate arrays with other increments, or more complicated indexing, you can gather the elements into a contiguous vector and then scatter them after the computation is complete.

VM uses the three following indexing methods to do this task:

- positive increment
- index vector

- mask vector

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in [Function Naming Conventions](#)). For more information on the indexing methods, see [Vector Arguments in VM](#) in Appendix B.

Error Diagnostics

The VM mathematical functions incorporate the error handling mechanism, which is controlled by the following service functions:

<code>vmlgeterrstatus</code> , <code>vmlseterrstatus</code> , <code>vmlclearerrstatus</code>	These functions operate with a global variable called VM Error Status. The VM Error Status flags an error, a warning, or a successful execution of a VM function.
<code>vmlgeterrcallback</code> , <code>vmlseterrcallback</code> , <code>vmlclearerrcallback</code>	These functions enable you to customize the error handling. For example, you can identify a particular argument in a vector where an error occurred or that caused a warning.
<code>vmlsetmode</code> , <code>vmlgetmode</code>	These functions get and set a VM mode. If you set a new VM mode using the <code>vmlsetmode</code> function, you can store the previous VM mode returned by the routine and restore it at any point of your application.

If both an error and a warning situation occur during the function call, the VM Error Status variable keeps only the value of the error code. See [Table "Values of the VM Error Status"](#) for possible values. If a VM function does not encounter errors or warnings, it sets the VM Error Status to `VML_STATUS_OK`.

If you use incorrect input arguments to a VM function (`VML_STATUS_BADSIZE` and `VML_STATUS_BADMEM`), the function calls `xerbla` to report errors. See [Table "Values of the VM Error Status"](#) for details.

You can use the `vmlsetmode` and `vmlgetmode` functions to modify error handling behavior. Depending on the VM mode, the error handling behavior includes the following operations:

- setting the VM Error Status to a value corresponding to the observed error or warning
- writing error text information to the `stderr` stream
- raising the appropriate exception on an error, if necessary
- calling the additional error handler callback function that is set by `vmlseterrorcallback`.

See Also

[vmlgeterrstatus](#) Gets the VM Error Status.

[vmlSetErrStatus](#) Sets the new VM Error Status according to `err` and stores the previous VM Error Status to `olderr`.

[vmlclearerrstatus](#) Sets the VM Error Status to `VML_STATUS_OK` and stores the previous VM Error Status to `olderr`.

[vmlSetErrorCallback](#) Sets the additional error handler callback function and gets the old callback function.

[vmlGetErrorCallback](#) Gets the additional error handler callback function.

[vmlClearErrorCallback](#) Deletes the additional error handler callback function and retrieves the former callback function.

[vmlgetmode](#) Gets the VM mode.

[vmlSetMode](#) Sets a new mode for VM functions according to the `mode` parameter and stores the previous VM mode to `oldmode`.

VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

Table "VM Mathematical Functions" lists available mathematical functions and associated data types.

VM Mathematical Functions

Function	Data Types	Description
Arithmetic Functions		
<code>v?add</code>	s, d, c, z	Addition of vector elements
<code>v?sub</code>	s, d, c, z	Subtraction of vector elements
<code>v?sqr</code>	s, d	Squaring of vector elements
<code>v?mul</code>	s, d, c, z	Multiplication of vector elements
<code>v?mulbyconj</code>	c, z	Multiplication of elements of one vector by conjugated elements of the second vector
<code>v?conj</code>	c, z	Conjugation of vector elements
<code>v?abs</code>	s, d, c, z	Computation of the absolute value of vector elements
<code>v?arg</code>	c, z	Computation of the argument of vector elements
<code>v?linearfrac</code>	s, d	Linear fraction transformation of vectors
Power and Root Functions		
<code>v?inv</code>	s, d	Inversion of vector elements
<code>v?div</code>	s, d, c, z	Division of elements of one vector by elements of the second vector
<code>v?sqrt</code>	s, d, c, z	Computation of the square root of vector elements
<code>v?invsqrt</code>	s, d	Computation of the inverse square root of vector elements
<code>v?cbrt</code>	s, d	Computation of the cube root of vector elements
<code>v?invcbrt</code>	s, d	Computation of the inverse cube root of vector elements
<code>v?pow2o3</code>	s, d	Raising each vector element to the power of 2/3
<code>v?pow3o2</code>	s, d	Raising each vector element to the power of 3/2
<code>v?pow</code>	s, d, c, z	Raising each vector element to the specified power
<code>v?powx</code>	s, d, c, z	Raising each vector element to the constant power
<code>v?hypot</code>	s, d	Computation of the square root of sum of squares
Exponential and Logarithmic Functions		
<code>v?exp</code>	s, d, c, z	Computation of the exponential of vector elements
<code>v?expm1</code>	s, d	Computation of the exponential of vector elements decreased by 1
<code>v?ln</code>	s, d, c, z	Computation of the natural logarithm of vector elements
<code>v?log10</code>	s, d, c, z	Computation of the denary logarithm of vector elements
<code>v?log1p</code>	s, d	Computation of the natural logarithm of vector elements that are increased by 1
Trigonometric Functions		
<code>v?cos</code>	s, d, c, z	Computation of the cosine of vector elements
<code>v?sin</code>	s, d, c, z	Computation of the sine of vector elements
<code>v?sincos</code>	s, d	Computation of the sine and cosine of vector elements
<code>v?cis</code>	c, z	Computation of the complex exponent of vector elements (cosine and sine combined to complex value)
<code>v?tan</code>	s, d, c, z	Computation of the tangent of vector elements
<code>v?acos</code>	s, d, c, z	Computation of the inverse cosine of vector elements

Function	Data Types	Description
<code>v?asin</code>	s, d, c, z	Computation of the inverse sine of vector elements
<code>v?atan</code>	s, d, c, z	Computation of the inverse tangent of vector elements
<code>v?atan2</code>	s, d	Computation of the four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
<code>v?cosh</code>	s, d, c, z	Computation of the hyperbolic cosine of vector elements
<code>v?sinh</code>	s, d, c, z	Computation of the hyperbolic sine of vector elements
<code>v?tanh</code>	s, d, c, z	Computation of the hyperbolic tangent of vector elements
<code>v?acosh</code>	s, d, c, z	Computation of the inverse hyperbolic cosine of vector elements
<code>v?asinh</code>	s, d, c, z	Computation of the inverse hyperbolic sine of vector elements
<code>v?atanh</code>	s, d, c, z	Computation of the inverse hyperbolic tangent of vector elements.
Special Functions		
<code>v?erf</code>	s, d	Computation of the error function value of vector elements
<code>v?erfc</code>	s, d	Computation of the complementary error function value of vector elements
<code>v?cdfnorm</code>	s, d	Computation of the cumulative normal distribution function value of vector elements
<code>v?erfinv</code>	s, d	Computation of the inverse error function value of vector elements
<code>v?erfcinv</code>	s, d	Computation of the inverse complementary error function value of vector elements
<code>v?cdfnorminv</code>	s, d	Computation of the inverse cumulative normal distribution function value of vector elements
<code>v?lgamma</code>	s, d	Computation of the natural logarithm for the absolute value of the gamma function of vector elements
<code>v?tgamma</code>	s, d	Computation of the gamma function of vector elements
Rounding Functions		
<code>v?floor</code>	s, d	Rounding towards minus infinity
<code>v?ceil</code>	s, d	Rounding towards plus infinity
<code>v?trunc</code>	s, d	Rounding towards zero infinity
<code>v?round</code>	s, d	Rounding to nearest integer
<code>v?nearbyint</code>	s, d	Rounding according to current mode
<code>v?rint</code>	s, d	Rounding according to current mode and raising inexact result exception
<code>v?modf</code>	s, d	Computation of the integer and fractional parts
<code>v?frac</code>	s, d	Computation of the fractional part

Special Value Notations

This section defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- z, z_1, z_2 , etc. denote complex numbers.
- $i, i^2=-1$ is the imaginary unit.
- x, X, x_1, x_2 , etc. denote real imaginary parts.
- y, Y, y_1, y_2 , etc. denote imaginary parts.
- X and Y represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet NaN and signaling NaN are denoted with `QNaN` and `SNAN`, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a + sign before X, Y , etc.
- The IEEE-754 negative infinities or floating-point numbers are denoted with a - sign before X, Y , etc.

`CONJ(z)` and `CIS(z)` are defined as follows:

$$\text{CONJ}(x+i\cdot y)=x-i\cdot y$$

$$\text{CIS}(y)=\cos(y)+i\cdot\sin(y).$$

The special value tables show the result of the function for the z argument at the intersection of the $\text{RE}(z)$ column and the $i * \text{IM}(z)$ row. If the function raises an exception on the argument z , the lower part of this cell shows the raised exception and the VM Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

v?Add

Performs element by element addition of vector a and vector b .

Syntax

```
call vsadd( n, a, b, y )
call vmsadd( n, a, b, y, mode )
call vdadd( n, a, b, y )
call vmdadd( n, a, b, y, mode )
call vcadd( n, a, b, y )
call vmcadd( n, a, b, y, mode )
call vzadd( n, a, b, y )
call vmzadd( n, a, b, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a, b	DOUBLE PRECISION for vdadd, vmdadd COMPLEX for vcadd, vmcadd DOUBLE COMPLEX for vzadd, vmzadd REAL, INTENT (IN) for vsAdd, vmsAdd DOUBLE PRECISION, INTENT (IN) for vdadd, vmdadd COMPLEX, INTENT (IN) for vcadd, vmcadd DOUBLE COMPLEX, INTENT (IN) for vzadd, vmzadd	Arrays that specify the input vectors a and b .

Name	Type	Description
<i>mode</i>	INTEGER (KIND=8) , INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vdadd</i> , <i>vmdadd</i> COMPLEX, for <i>vcadd</i> , <i>vmcadd</i> DOUBLE COMPLEX for <i>vzadd</i> , <i>vmzadd</i> REAL, INTENT (OUT) for <i>vsAdd</i> , <i>vmsAdd</i> DOUBLE PRECISION, INTENT (OUT) for <i>vdadd</i> , <i>vmdadd</i> COMPLEX, INTENT (OUT) for <i>vcadd</i> , <i>vmcadd</i> DOUBLE COMPLEX, INTENT (OUT) for <i>vzadd</i> , <i>vmzadd</i>	Array that specifies the output vector <i>y</i> .

Description

The *v?Add* function performs element by element addition of vector *a* and vector *b*.

Special values for Real Function *v?Add(x)*

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
$+\infty$	$+\infty$	$+\infty$	
$+\infty$	$-\infty$	QNAN	INVALID
$-\infty$	$+\infty$	QNAN	INVALID
$-\infty$	$-\infty$	$-\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Add}(x_1+i*y_1, x_2+i*y_2) = (x_1+x_2) + i*(y_1+y_2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when x_1 , x_2 , y_1 , y_2 are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Sub

Performs element by element subtraction of vector *b* from vector *a*.

Syntax

```
call vssub( n, a, b, y )
call vmssub( n, a, b, y, mode )
call vdsusb( n, a, b, y )
call vmdsusb( n, a, b, y, mode )
call vcsusb( n, a, b, y )
call vmcsusb( n, a, b, y, mode )
call vzsub( n, a, b, y )
call vmzsub( n, a, b, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a, b</i>	DOUBLE PRECISION for vdsusb, vmdsusb COMPLEX for vcsusb, vmcsusb DOUBLE COMPLEX for vzsub, vmzsub REAL, INTENT (IN) for vssub, vmssub DOUBLE PRECISION, INTENT (IN) for vdsusb, vmdsusb COMPLEX, INTENT (IN) for vcsusb, vmcsusb DOUBLE COMPLEX, INTENT (IN) for vzsub, vmzsub	Arrays that specify the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vssub, vmssub DOUBLE PRECISION for vdsusb, vmdsusb	Array that specifies the output vector <i>y</i> .

Name	Type	Description
	COMPLEX	for <code>vcsb</code> , <code>vmcsb</code>
	DOUBLE COMPLEX	for <code>vzsb</code> , <code>vmzsb</code>
	REALINTENT(OUT)	for <code>vssb</code> , <code>vmssb</code>
	DOUBLE PRECISION, INTENT(OUT)	for <code>vdsb</code> , <code>vmdsb</code>
	COMPLEX, INTENT(OUT)	for <code>vcsb</code> , <code>vmcsb</code>
	DOUBLE COMPLEX, INTENT(OUT)	for <code>vzsb</code> , <code>vmzsb</code>

Description

The `v?Sub` function performs element by element subtraction of vector *b* from vector *a*.

Special values for Real Function `v?Sub(x)`

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	-0	
-0	-0	+0	
+∞	+∞	QNAN	INVALID
+∞	-∞	+∞	
-∞	+∞	-∞	
-∞	-∞	QNAN	INVALID
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sub}(x1+i*y1, x2+i*y2) = (x1-x2) + i*(y1-y2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when *x1*, *x2*, *y1*, *y2* are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

`v?Sqr`

Performs element by element squaring of the vector.

Syntax

```
call vssqr( n, a, y )
call vmssqr( n, a, y, mode )
call vdsqr( n, a, y )
call vmdsqr( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdsqr</code> , <code>vmdsqr</code> REAL, INTENT (IN) for <code>vssqr</code> , <code>vmssqr</code> DOUBLE PRECISION, INTENT (IN) for <code>vdsqr</code> , <code>vmdsqr</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdsqr</code> , <code>vmdsqr</code> REAL, INTENT (OUT) for <code>vssqr</code> , <code>vmssqr</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdsqr</code> , <code>vmdsqr</code>	Array that specifies the output vector <i>y</i> .

Description

The `v?Sqr` function performs element by element squaring of the vector.

Special Values for Real Function `v?Sqr(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

`v?Mul`

*Performs element by element multiplication of vector *a* and vector *b*.*

Syntax

```
call vsmul( n, a, b, y )
call vmsmul( n, a, b, y, mode )
call vdmul( n, a, b, y )
call vmdmul( n, a, b, y, mode )
```

```
call vcmul( n, a, b, y )
call vmcmul( n, a, b, y, mode )
call vzmul( n, a, b, y )
call vmzmul( n, a, b, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a, b</i>	DOUBLE PRECISION for <code>vdmul</code> , <code>vmdmul</code> COMPLEX for <code>vcmul</code> , <code>vmcmul</code> DOUBLE COMPLEX for <code>vzmul</code> , <code>vmzmul</code> REAL, INTENT(IN) for <code>vsmul</code> , <code>vmsmul</code> DOUBLE PRECISION, INTENT(IN) for <code>vdmul</code> , <code>vmdmul</code> COMPLEX, INTENT(IN) for <code>vcmul</code> , <code>vmcmul</code> DOUBLE COMPLEX, INTENT(IN) for <code>vzmul</code> , <code>vmzmul</code>	Arrays that specify the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdmul</code> , <code>vmdmul</code> COMPLEX, for <code>vcmul</code> , <code>vmcmul</code> DOUBLE COMPLEX for <code>vzmul</code> , <code>vmzmul</code> REAL, INTENT(OUT) for <code>vsmul</code> , <code>vmsmul</code> DOUBLE PRECISION, INTENT(OUT) for <code>vdmul</code> , <code>vmdmul</code> COMPLEX, INTENT(OUT) for <code>vcmul</code> , <code>vmcmul</code> DOUBLE COMPLEX, INTENT(OUT) for <code>vzmul</code> , <code>vmzmul</code>	Array that specifies the output vector <i>y</i> .

Description

The `v?Mul` function performs element by element multiplication of vector a and vector b .

Special values for Real Function `v?Mul(x)`

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	$+\infty$	QNAN	INVALID
+0	$-\infty$	QNAN	INVALID
-0	$+\infty$	QNAN	INVALID
-0	$-\infty$	QNAN	INVALID
$+\infty$	+0	QNAN	INVALID
$+\infty$	-0	QNAN	INVALID
$-\infty$	+0	QNAN	INVALID
$-\infty$	-0	QNAN	INVALID
$+\infty$	$+\infty$	$+\infty$	
$+\infty$	$-\infty$	$-\infty$	
$-\infty$	$+\infty$	$-\infty$	
$-\infty$	$-\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Mul}(x1+i*y1, x2+i*y2) = (x1*x2-y1*y2) + i*(x1*y2+y1*x2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when $x1$, $x2$, $y1$, $y2$ are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

`v?MulByConj`

Performs element by element multiplication of vector a element and conjugated vector b element.

Syntax

```
call vcmulbyconj( n, a, b, y )
call vmcmulbyconj( n, a, b, y, mode )
call vzmulbyconj( n, a, b, y )
call vmzmulbyconj( n, a, b, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a, b	DOUBLE COMPLEX for <code>vzmulbyconj</code> , <code>vmzmulbyconj</code> COMPLEX, INTENT (IN) for <code>vcmulbyconj</code> , <code>vmcmulbyconj</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzmulbyconj</code> , <code>vmzmulbyconj</code>	Arrays that specify the input vectors a and b .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE COMPLEX for <code>vzmulbyconj</code> , <code>vmzmulbyconj</code> COMPLEX, INTENT (OUT) for <code>vcmulbyconj</code> , <code>vmcmulbyconj</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzmulbyconj</code> , <code>vmzmulbyconj</code>	Array that specifies the output vector y .

Description

The `v?MulByConj` function performs element by element multiplication of vector a element and conjugated vector b element.

Specifications for special values of the functions are found according to the formula

$$\text{MulByConj}(x1+i*y1, x2+i*y2) = \text{Mul}(x1+i*y1, x2-i*y2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when $x1, x2, y1, y2$ are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Conj

Performs element by element conjugation of the vector.

Syntax

```
call vcconj( n, a, y )
call vmcconj( n, a, y, mode )
call vzconj( n, a, y )
call vmzconj( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE COMPLEX, INTENT (IN) for <code>vzconj</code> , <code>vmzconj</code> COMPLEX, INTENT (IN) for <code>vcconj</code> , <code>vmcconj</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzconj</code> , <code>vmzconj</code>	Array that specifies the input vector <code>a</code> .
<code>mode</code>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	DOUBLE COMPLEX for <code>vzconj</code> , <code>vmzconj</code> COMPLEX, INTENT (OUT) for <code>vcconj</code> , <code>vmcconj</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzconj</code> , <code>vmzconj</code>	Array that specifies the output vector <code>y</code> .

Description

The `v?Conj` function performs element by element conjugation of the vector.

No special values are specified. The function does not raise floating-point exceptions.

v?Abs

Computes absolute value of vector elements.

Syntax

```
call vsabs( n, a, y )
call vmsabs( n, a, y, mode )
call vdabs( n, a, y )
call vmdabs( n, a, y, mode )
call vcabs( n, a, y )
call vmcabs( n, a, y, mode )
call vzabs( n, a, y )
call vmzabs( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for <code>vdabs</code> , <code>vmdabs</code> COMPLEX for <code>vcabs</code> , <code>vmcabs</code> DOUBLE COMPLEX for <code>vzabs</code> , <code>vmzabs</code> REAL, INTENT (IN) for <code>vsabs</code> , <code>vmsabs</code> DOUBLE PRECISION, INTENT (IN) for <code>vdabs</code> , <code>vmdabs</code> COMPLEX, INTENT (IN) for <code>vcabs</code> , <code>vmcabs</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzabs</code> , <code>vmzabs</code>	Array that specifies the input vector a .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdabs</code> , <code>vmdabs</code> , <code>vzabs</code> , <code>vmzabs</code> REAL, INTENT (OUT) for <code>vsabs</code> , <code>vmsabs</code> , <code>vcabs</code> , <code>vmcabs</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdabs</code> , <code>vmdabs</code> , <code>vzabs</code> , <code>vmzabs</code>	Array that specifies the output vector y .

Description

The `v?Abs` function computes an absolute value of vector elements.

Special Values for Real Function `v?Abs(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Abs}(z) = \text{Hypot}(\text{RE}(z), \text{IM}(z)).$$

v?Arg

Computes argument of vector elements.

Syntax

```
call vcarg( n, a, y )
call vmcarg( n, a, y, mode )
call vzarg( n, a, y )
call vmzarg( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN77: COMPLEX for <i>vcarg</i> , <i>vmcarg</i> DOUBLE COMPLEX for <i>vzarg</i> , <i>vmzarg</i> COMPLEX, INTENT (IN) for <i>vcarg</i> , <i>vmcarg</i> DOUBLE COMPLEX, INTENT (IN) for <i>vzarg</i> , <i>vmzarg</i>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vzarg</i> , <i>vmzarg</i> REAL, INTENT (OUT) for <i>vcarg</i> , <i>vmcarg</i> DOUBLE PRECISION, INTENT (OUT) for <i>vzarg</i> , <i>vmzarg</i>	Array that specifies the output vector <i>y</i> .

Description

The *v?Arg* function computes argument of vector elements.

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Arg(z)$

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+3\cdot\pi/4$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4$	NAN
$+i\cdot Y$	$+\pi$		$+\pi/2$	$+\pi/2$		$+0$	NAN
$+i\cdot 0$	$+\pi$	$+\pi$	$+\pi$	$+0$	$+0$	$+0$	NAN
$-i\cdot 0$	$-\pi$	$-\pi$	$-\pi$	-0	-0	-0	NAN
$-i\cdot Y$	$-\pi$		$-\pi/2$	$-\pi/2$		-0	NAN
$-i\cdot\infty$	$-3\cdot\pi/4$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4$	NAN
$+i\cdot\text{NAN}$	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $Arg(z) = \text{Atan2}(IM(z), RE(z))$.

$v?LinearFrac$

Performs linear fraction transformation of vectors *a* and *b* with scalar parameters.

Syntax

```
call vslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
call vmslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
call vdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
call vmdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a, b</i>	DOUBLE PRECISION for <code>vdlinearfrac, vmdlinearfrac</code> REAL, INTENT(IN) for <code>vslinearfrac, vmslinearfrac</code> DOUBLE PRECISION, INTENT(IN) for <code>vdlinearfrac, vmdlinearfrac</code>	Arrays that specify the input vectors <i>a</i> and <i>b</i> .
<i>scalea, scaleb</i>	DOUBLE PRECISION for <code>vdlinearfrac, vmdlinearfrac</code> REAL, INTENT(IN) for <code>vslinearfrac, vmslinearfrac</code>	Constant values for shifting addends of vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac	
<i>shifta, shiftb</i>	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac	Constant values for scaling multipliers of vectors <i>a</i> and <i>b</i> .
	REAL, INTENT(IN) for vslinearfrac, vmslinearfrac	
	DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac	
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac	Array that specifies the output vector <i>y</i> .
	REAL, INTENT(OUT) for vslinearfrac, vmslinearfrac	
	DOUBLE PRECISION, INTENT(OUT) for vdlinearfrac, vmdlinearfrac	

Description

The `v?LinearFrac` function performs linear fraction transformation of vectors *a* by vector *b* with scalar parameters: scaling multipliers *scalea*, *scaleb* and shifting addends *shifta*, *shiftb*:

$$y[i] = (scalea \cdot a[i] + shifta) / (scaleb \cdot b[i] + shiftb), i=1, 2 \dots n$$

The `v?LinearFrac` function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, `v?LinearFrac` sets the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VM Status](#) table). Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters

$$2^{E_{MIN}/2} \leq |scalea| \leq 2^{(E_{MAX}-2)/2}$$

$$2^{E_{MIN}/2} \leq |scaleb| \leq 2^{(E_{MAX}-2)/2}$$

$$|shifta| \leq 2^{E_{MAX}-2}$$

$$|shiftb| \leq 2^{E_{MAX}-2}$$

$$2^{E_{MIN}/2} \leq a[i] \leq 2^{(E_{MAX}-2)/2}$$

$$2^{E_{MIN}/2} \leq b[i] \leq 2^{(E_{MAX}-2)/2}$$

$$a[i] \neq - (shifta/scalea) * (1-\delta_1), \quad |\delta_1| \leq 2^{1-(p-1)/2}$$

$$b[i] \neq - (shiftb/scaleb) * (1-\delta_2), \quad |\delta_2| \leq 2^{1-(p-1)/2}$$

E_{MIN} and E_{MAX} are the maximum and minimum exponents and *p* is the number of significant bits (precision) for corresponding data type according to the ANSI/IEEE Std 754-2008 standard ([IEEE754]):

- for single precision $E_{\text{MIN}} = -126$, $E_{\text{MAX}} = 127$, $p = 24$
- for double precision $E_{\text{MIN}} = -1022$, $E_{\text{MAX}} = 1023$, $p = 53$

The thresholds become less strict for common cases with $scalea=0$ and/or $scaleb=0$:

- if $scalea=0$, there are no limitations for the values of $a[i]$ and $shifta$
- if $scaleb=0$, there are no limitations for the values of $b[i]$ and $shiftb$

Power and Root Functions

v?Inv

Performs element by element inversion of the vector.

Syntax

```
call vsinv( n, a, y )
call vmsinv( n, a, y, mode )
call vdiv( n, a, y )
call vmdinv( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdiv, vmdinv REAL, INTENT (IN) for vsinv, vmsinv DOUBLE PRECISION, INTENT (IN) for vdiv, vmdinv	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdiv, vmdinv REAL, INTENT (OUT) for vsinv, vmsinv DOUBLE PRECISION, INTENT (OUT) for vdiv, vmdinv	Array that specifies the output vector <i>y</i> .

Description

The `v?Inv` function performs element by element inversion of the vector.

Special Values for Real Function `v?Inv(x)`

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$+\infty$	+0		
$-\infty$	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Div`

Performs element by element division of vector *a* by vector *b*

Syntax

```
call vsdiv( n, a, b, y )
call vmsdiv( n, a, b, y, mode )
call vddiv( n, a, b, y )
call vmddiv( n, a, b, y, mode )
call vcdiv( n, a, b, y )
call vmcdiv( n, a, b, y, mode )
call vzdiv( n, a, b, y )
call vmzdiv( n, a, b, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a, b</i>	DOUBLE PRECISION for <code>vddiv</code> , <code>vmddiv</code> COMPLEX for <code>vcdiv</code> , <code>vmcdiv</code> DOUBLE COMPLEX for <code>vzdiv</code> , <code>vmzdiv</code> REAL, INTENT(IN) for <code>vsdiv</code> , <code>vmsdiv</code> DOUBLE PRECISION, INTENT(IN) for <code>vddiv</code> , <code>vmddiv</code> COMPLEX, INTENT(IN) for <code>vcdiv</code> , <code>vmcdiv</code>	Arrays that specify the input vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vzdiv, vmzdiv	
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Div Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{FLT_MAX}$
double precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{DBL_MAX}$

Precision overflow thresholds for the complex v?Div function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vddiv, vmddiv COMPLEX for vcdiv, vmcdiv DOUBLE COMPLEX for vzdiv, vmzdiv REAL, INTENT(OUT) for vsdiv, vmsdiv DOUBLE PRECISION, INTENT(OUT) for vddiv, vmddiv COMPLEX, INTENT(OUT) for vcdiv, vmcdiv DOUBLE COMPLEX, INTENT(OUT) for vzdiv, vmzdiv	Array that specifies the output vector <i>y</i> .

Description

The v?Div function performs element by element division of vector *a* by vector *b*.

Special values for Real Function v?Div(x)

Argument 1	Argument 2	Result	VM Error Status	Exception
X > +0	+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
X > +0	-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
X < +0	+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
X < +0	-0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
+0	+0	QNAN	VML_STATUS_SING	
-0	-0	QNAN	VML_STATUS_SING	
X > +0	$+\infty$	+0		
X > +0	$-\infty$	-0		
$+\infty$	$+\infty$	QNAN		
$-\infty$	$-\infty$	QNAN		
QNAN	QNAN	QNAN		
SNAN	SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Div}(x1+i*y1, x2+i*y2) = (x1+i*y1)*(x2-i*y2) / (x2*x2+y2*y2).$$

Overflow in a complex function occurs when $x2+i*y2$ is not zero, $x1, x2, y1, y2$ are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

v?Sqrt

Computes a square root of vector elements.

Syntax

```
call vssqrt( n, a, y )
call vmssqrt( n, a, y, mode )
call vdsqrt( n, a, y )
call vmdsqrt( n, a, y, mode )
call vcsqrt( n, a, y )
call vmcsqrt( n, a, y, mode )
call vzsqrt( n, a, y )
call vmzsqrt( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdsqrt</code> , <code>vmdsqrt</code> COMPLEX for <code>vcsqrt</code> , <code>vmcsqrt</code> DOUBLE COMPLEX for <code>vzsqrt</code> , <code>vmzsqrt</code> REAL, INTENT (IN) for <code>vssqrt</code> , <code>vmssqrt</code> DOUBLE PRECISION, INTENT (IN) for <code>vdsqrt</code> , <code>vmdsqrt</code> COMPLEX, INTENT (IN) for <code>vcsqrt</code> , <code>vmcsqrt</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzsqrt</code> , <code>vmzsqrt</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
y	REAL for <code>vssqrt</code> , <code>vmssqrt</code> DOUBLE PRECISION for <code>vdsqrt</code> , <code>vmdsqrt</code> COMPLEX for <code>vcsqrt</code> , <code>vmcsqrt</code> DOUBLE COMPLEX for <code>vzsqrt</code> , <code>vmzsqrt</code> REAL, INTENT(OUT) for <code>vssqrt</code> , <code>vmssqrt</code> DOUBLE PRECISION, INTENT(OUT) for <code>vdsqrt</code> , <code>vmdsqrt</code> COMPLEX, INTENT(OUT) for <code>vcsqrt</code> , <code>vmcsqrt</code> DOUBLE COMPLEX, INTENT(OUT) for <code>vzsqrt</code> , <code>vmzsqrt</code>	Array that specifies the output vector y .

Description

The `v?Sqrt` function computes a square root of vector elements.

Special Values for Real Function `v?Sqrt(x)`

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
$+0$	$+0$		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Sqrt(z)`

RE(z) i·IM(z)	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$
$+i\cdot Y$	$+0+i\cdot\infty$					$+\infty+i\cdot 0$	QNAN+i·QNAN
$+i\cdot 0$	$+0+i\cdot\infty$		$+0+i\cdot 0$	$+0+i\cdot 0$		$+\infty+i\cdot 0$	QNAN+i·QNAN
$-i\cdot 0$	$+0-i\cdot\infty$		$+0-i\cdot 0$	$+0-i\cdot 0$		$+\infty-i\cdot 0$	QNAN+i·QNAN
$-i\cdot Y$	$+0-i\cdot\infty$					$+\infty-i\cdot 0$	QNAN+i·QNAN
$-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$	$+\infty-i\cdot\infty$
$+i\cdot\text{NAN}$	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	$+\infty+i\cdot\text{QNAN}$	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- $\text{Sqrt}(\text{CONJ}(z)) = \text{CONJ}(\text{Sqrt}(z))$.

v?InvSqrt

Computes an inverse square root of vector elements.

Syntax

```
call vsinvsqrt( n, a, y )
```

```
call vmsinvsqrt( n, a, y, mode )
```

```
call vdinvsqrt( n, a, y )
```

```
call vmdinvsqrt( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdinvsqrt</code> , <code>vmdinvsqrt</code> REAL, INTENT(IN) for <code>vsinvsqrt</code> , <code>vmsinvsqrt</code> DOUBLE PRECISION, INTENT(IN) for <code>vdinvsqrt</code> , <code>vmdinvsqrt</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdinvsqrt</code> , <code>vmdinvsqrt</code> REAL, INTENT(OUT) for <code>vsinvsqrt</code> , <code>vmsinvsqrt</code> DOUBLE PRECISION, INTENT(OUT) for <code>vdinvsqrt</code> , <code>vmdinvsqrt</code>	Array that specifies the output vector <i>y</i> .

Description

The `v?InvSqrt` function computes an inverse square root of vector elements.

Special Values for Real Function v?InvSqrt(x)

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Cbrt

Computes a cube root of vector elements.

Syntax

```
call vsqrt( n, a, y )
call vmsqrt( n, a, y, mode )
call vdcbrt( n, a, y )
call vmdcbrt( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdcbrt, vmdcbrt REAL, INTENT(IN) for vsqrt, vmsqrt DOUBLE PRECISION, INTENT(IN) for vdcbrt, vmdcbrt	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdcbrt, vmdcbrt REAL, INTENT(OUT) for vsqrt, vmsqrt DOUBLE PRECISION, INTENT(OUT) for vdcbrt, vmdcbrt	Array that specifies the output vector <i>y</i> .

Description

The `v?Cbrt` function computes a cube root of vector elements.

Special Values for Real Function `v?Cbrt(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

`v?InvCbrt`

Computes an inverse cube root of vector elements.

Syntax

```
call vsinvcbrt( n, a, y )
```

```
call vmsinvcbrt( n, a, y, mode )
```

```
call vdinvcbrt( n, a, y )
```

```
call vmdinvcbrt( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdinvcbrt</code> , <code>vmdinvcbrt</code> REAL, INTENT (IN) for <code>vsinvcbrt</code> , <code>vmsinvcbrt</code> DOUBLE PRECISION, INTENT (IN) for <code>vdinvcbrt</code> , <code>vmdinvcbrt</code>	Array that specifies the input vector <code>a</code> .
<code>mode</code>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	DOUBLE PRECISION for <code>vdinvcbrt</code> , <code>vmdinvcbrt</code> REAL, INTENT (OUT) for <code>vsinvcbrt</code> , <code>vmsinvcbrt</code>	Array that specifies the output vector <code>y</code> .

Name	Type	Description
------	------	-------------

	DOUBLE PRECISION, INTENT(OUT) for vdinvcbrt, vmdinvcbrt	
--	--	--

Description

The `v?InvCbrt` function computes an inverse cube root of vector elements.

Special Values for Real Function `v?InvCbrt(x)`

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$+\infty$	+0		
$-\infty$	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Pow2o3`

Raises each element of a vector to the constant power 2/3.

Syntax

```
call vspow2o3( n, a, y )
call vmspow2o3( n, a, y, mode )
call vdpow2o3( n, a, y )
call vmdpow2o3( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdpow2o3</code> , <code>vmdpow2o3</code> REAL, INTENT(IN) for <code>vspow2o3</code> , <code>vmspow2o3</code> DOUBLE PRECISION, INTENT(IN) for <code>vdpow2o3</code> , <code>vmdpow2o3</code>	Arrays, specify the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdpow2o3</code> , <code>vmdpov2o3</code> REAL, INTENT (OUT) for <code>vspow2o3</code> , <code>vmspow2o3</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdpow2o3</code> , <code>vmdpov2o3</code>	Array, specifies the output vector y .

Description

The `v?Pow2o3` function raises each element of a vector to the constant power $2/3$.

Special Values for Real Function `v?Pow2o3(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNaN	QNaN	
SNAN	QNaN	INVALID

`v?Pow3o2`

Raises each element of a vector to the constant power $3/2$.

Syntax

```
call vspow3o2( n, a, y )
call vmspow3o2( n, a, y, mode )
call vdpow3o2( n, a, y )
call vmdpov3o2( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for <code>vdpow3o2</code> , <code>vmdpov3o2</code> REAL, INTENT (IN) for <code>vspow3o2</code> , <code>vmspow3o2</code> DOUBLE PRECISION, INTENT (IN) for <code>vdpow3o2</code> , <code>vmdpov3o2</code>	Arrays, specify the input vector a .

Name	Type	Description
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Pow3o2 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{2/3}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{2/3}$

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdpow3o2</code> , <code>vmdpow3o2</code> REAL, INTENT (OUT) for <code>vspow3o2</code> , <code>vmispow3o2</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdipow3o2</code> , <code>vmdipow3o2</code>	Array, specifies the output vector <i>y</i> .

Description

The `v?Pow3o2` function raises each element of a vector to the constant power 3/2.

Special Values for Real Function `v?Pow3o2(x)`

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Pow`

Computes *a* to the power *b* for elements of two vectors.

Syntax

```
call vspow( n, a, b, y )
call vmispow( n, a, b, y, mode )
call vdpow( n, a, b, y )
call vmdpow( n, a, b, y, mode )
call vcpow( n, a, b, y )
call vmcpow( n, a, b, y, mode )
call vzipow( n, a, b, y )
call vmzipow( n, a, b, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a, b</i>	DOUBLE PRECISION for <code>vdpow</code> , <code>vmdpaw</code> COMPLEX for <code>vcpow</code> , <code>vmcpaw</code> DOUBLE COMPLEX for <code>vzpow</code> , <code>vmzpow</code> REAL, INTENT (IN) for <code>vspow</code> , <code>vmspow</code> DOUBLE PRECISION, INTENT (IN) for <code>vdpow</code> , <code>vmdpaw</code> COMPLEX, INTENT (IN) for <code>vcpow</code> , <code>vmcpaw</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzpow</code> , <code>vmzpow</code>	Arrays that specify the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Pow Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{1/b[i]}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{1/b[i]}$

Precision overflow thresholds for the complex `v?Pow` function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdpow</code> , <code>vmdpaw</code> COMPLEX for <code>vcpow</code> , <code>vmcpaw</code> DOUBLE COMPLEX for <code>vzpow</code> , <code>vmzpow</code> REAL, INTENT (OUT) for <code>vspow</code> , <code>vmspow</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdpow</code> , <code>vmdpaw</code> COMPLEX, INTENT (OUT) for <code>vcpow</code> , <code>vmcpaw</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzpow</code> , <code>vmzpow</code>	Array that specifies the output vector <i>y</i> .

Description

The $v?Pow$ function computes a to the power b for elements of two vectors.

The real function $v(s/d)Pow$ has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then $b[i]$ may be arbitrary. For negative $a[i]$, the value of $b[i]$ must be an integer (either positive or negative).

The complex function $v(c/z)Pow$ has no input range limitations.

Special values for Real Function $v?Pow(x)$

Argument 1	Argument 2	Result	VM Error Status	Exception
+0	neg. odd integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. odd integer	$-\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	pos. odd integer	+0		
-0	pos. odd integer	-0		
+0	pos. even integer	+0		
-0	pos. even integer	+0		
+0	pos. non-integer	+0		
-0	pos. non-integer	+0		
-1	$+\infty$	+1		
-1	$-\infty$	+1		
+1	any value	+1		
+1	+0	+1		
+1	-0	+1		
+1	$+\infty$	+1		
+1	$-\infty$	+1		
+1	QNAN	+1		
any value	+0	+1		
+0	+0	+1		
-0	+0	+1		
$+\infty$	+0	+1		
$-\infty$	+0	+1		
QNAN	+0	+1		
any value	-0	+1		
+0	-0	+1		
-0	-0	+1		
$+\infty$	-0	+1		
$-\infty$	-0	+1		
QNAN	-0	+1		
$X < +0$	non-integer	QNAN	VML_STATUS_ERRDOM	INVALID
$ X < 1$	$-\infty$	$+\infty$		
+0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
$ X > 1$	$-\infty$	+0		
$+\infty$	$-\infty$	+0		
$-\infty$	$-\infty$	+0		

Argument 1	Argument 2	Result	VM Error Status	Exception
$ X < 1$	$+\infty$	$+0$		
$+0$	$+\infty$	$+0$		
-0	$+\infty$	$+0$		
$ X > 1$	$+\infty$	$+\infty$		
$+\infty$	$+\infty$	$+\infty$		
$-\infty$	$+\infty$	$+\infty$		
$-\infty$	neg. odd integer	-0		
$-\infty$	neg. even integer	$+0$		
$-\infty$	neg. non-integer	$+0$		
$-\infty$	pos. odd integer	$-\infty$		
$-\infty$	pos. even integer	$+\infty$		
$-\infty$	pos. non-integer	$+\infty$		
$+\infty$	$X < +0$	$+0$		
$+\infty$	$X > +0$	$+\infty$		
QNAN	QNAN	QNAN		
QNAN	SNAN	QNAN		INVALID
SNAN	QNAN	QNAN		INVALID
SNAN	SNAN	QNAN		INVALID

The complex double precision versions of this function, `vzPow` and `vmzPow`, are implemented in the EP accuracy mode only. If used in HA or LA mode, `vzPow` and `vmzPow` set the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VM Status table](#)).

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when x_1 , x_2 , y_1 , y_2 are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Powx

Raises each element of a vector to the constant power.

Syntax

```
call vspowx( n, a, b, y )
call vmspowx( n, a, b, y, mode )
call vdpowx( n, a, b, y )
call vmdpowx( n, a, b, y, mode )
call vcpowx( n, a, b, y )
call vmcpowx( n, a, b, y, mode )
call vzpowx( n, a, b, y )
call vmzpowx( n, a, b, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Number of elements to be calculated.
a	DOUBLE PRECISION for <code>vdpowx</code> , <code>vmdpowx</code> COMPLEX for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX for <code>vzpowx</code> , <code>vmzpowx</code> REAL, INTENT (IN) for <code>vspowx</code> , <code>vmspowx</code> DOUBLE PRECISION, INTENT (IN) for <code>vdpowx</code> , <code>vmdpowx</code> COMPLEX, INTENT (IN) for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzpowx</code> , <code>vmzpowx</code>	Array a that specifies the input vector
b	DOUBLE PRECISION for <code>vdpowx</code> , <code>vmdpowx</code> COMPLEX for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX for <code>vzpowx</code> , <code>vmzpowx</code> REAL, INTENT (IN) for <code>vspowx</code> , <code>vmspowx</code> DOUBLE PRECISION, INTENT (IN) for <code>vdpowx</code> , <code>vmdpowx</code> COMPLEX, INTENT (IN) for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX, INTENT (IN) for <code>vzpowx</code> , <code>vmzpowx</code>	Scalar value b that is the constant power.
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real `v?Powx` Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{1/b}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{1/b}$

Precision overflow thresholds for the complex `v?Powx` function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdpowx</code> , <code>vmdpox</code> COMPLEX for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX for <code>vzpowx</code> , <code>vmzpowx</code> REAL, INTENT (OUT) for <code>vspowx</code> , <code>vmspox</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdpowx</code> , <code>vmdpox</code> COMPLEX, INTENT (OUT) for <code>vcpowx</code> , <code>vmcpowx</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzpowx</code> , <code>vmzpowx</code>	Array that specifies the output vector y .

Description

The `v?Powx` function raises each element of a vector to the constant power.

The real function `v(s/d)Powx` has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then b may be arbitrary. For negative $a[i]$, the value of b must be an integer (either positive or negative).

The complex function `v(c/z)Powx` has no input range limitations.

Special values and VM Error Status treatment are the same as for the `v?Pow` function.

v?Hypot

Computes a square root of sum of two squared elements.

Syntax

```
call vshypot( n, a, b, y )
call vmshypot( n, a, b, y, mode )
call vdhypot( n, a, b, y )
call vmdhypot( n, a, b, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Number of elements to be calculated.
a, b	DOUBLE PRECISION for <code>vdhypot</code> , <code>vmdhypot</code>	Arrays that specify the input vectors a and b

Name	Type	Description
	REAL, INTENT(IN) for vshypot, vmshypot DOUBLE PRECISION, INTENT(IN) for vdhypot, vmdhypot	
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Hypot Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{sqrt}(\text{FLT_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{FLT_MAX})$
double precision	$\text{abs}(a[i]) < \text{sqrt}(\text{DBL_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{DBL_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdhypot, vmdhypot REAL, INTENT(OUT) for vshypot, vmshypot DOUBLE PRECISION, INTENT(OUT) for vdhypot, vmdhypot	Array that specifies the output vector <i>y</i> .

Description

The function `v?Hypot` computes a square root of sum of two squared elements.

Special values for Real Function `v?Hypot(x)`

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
-0	-0	+0	
$+\infty$	any value	$+\infty$	
any value	$+\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Exponential and Logarithmic Functions

v?Exp

Computes an exponential of vector elements.

Syntax

```
call vsexp( n, a, y )
call vmsexp( n, a, y, mode )
call vdexp( n, a, y )
call vmdexp( n, a, y, mode )
call vcexp( n, a, y )
call vmcexp( n, a, y, mode )
call vzexp( n, a, y )
call vmzexp( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdexp, vmdexp COMPLEX for vcexp, vmcexp DOUBLE COMPLEX for vzexp, vmzexp REAL, INTENT (IN) for vsexp, vmsexp DOUBLE PRECISION, INTENT (IN) for vdexp, vmdexp COMPLEX, INTENT (IN) for vcexp, vmcexp DOUBLE COMPLEX, INTENT (IN) for vzexp, vmzexp	Array, specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Exp Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \text{Ln}(\text{FLT_MAX})$
double precision	$a[i] < \text{Ln}(\text{DBL_MAX})$

Precision overflow thresholds for the complex v?Exp function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for $vdexp$, $vmdexp$ COMPLEX for $vcexp$, $vmcexp$ DOUBLE COMPLEX for $vzexp$, $vmzexp$ REAL, INTENT (OUT) for $vsexp$, $vmsexp$ DOUBLE PRECISION, INTENT (OUT) for $vdexp$, $vmdexp$ COMPLEX, INTENT (OUT) for $vcexp$, $vmcexp$ DOUBLE COMPLEX, INTENT (OUT) for $vzexp$, $vmzexp$	Array, specifies the output vector y .

Description

The $v?Exp$ function computes an exponential of vector elements.

Special Values for Real Function $v?Exp(x)$

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$X >$ overflow	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$X <$ underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	$+\infty$		
$-\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Exp(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i \cdot \infty$	$+0+i \cdot 0$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i \cdot$ QNAN INVALID	QNAN+i·QNAN INVALID
$+i \cdot Y$	$+0 \cdot CIS(Y)$					$+\infty \cdot CIS(Y)$	QNAN+i·QNAN
$+i \cdot 0$	$+0 \cdot CIS(0)$		$+1+i \cdot 0$	$+1+i \cdot 0$		$+\infty+i \cdot 0$	QNAN+i·0
$-i \cdot 0$	$+0 \cdot CIS(0)$		$+1-i \cdot 0$	$+1-i \cdot 0$		$+\infty-i \cdot 0$	QNAN-i·0
$-i \cdot Y$	$+0 \cdot CIS(Y)$					$+\infty \cdot CIS(Y)$	QNAN+i·QNAN
$-i \cdot \infty$	$+0-i \cdot 0$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i \cdot$ QNAN INVALID	QNAN+i·QNAN
$+i \cdot NAN$	$+0+i \cdot 0$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i \cdot$ QNAN	QNAN+i·QNAN

Notes:

- raises the `INVALID` exception when real or imaginary part of the argument is `SNAN`
- raises the `INVALID` exception on argument $z = -\infty + i \cdot \text{QNAN}$
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when $\text{RE}(z)$, $\text{IM}(z)$ are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

v?Expm1

Computes an exponential of vector elements decreased by 1.

Syntax

```
call vsexpm1( n, a, y )
call vmsexpm1( n, a, y, mode )
call vdexpm1( n, a, y )
call vdexpm1( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdexpm1</code> , <code>vmsexpm1</code> REAL, INTENT (IN) for <code>vsexpm1</code> , <code>vmsexpm1</code> DOUBLE PRECISION, INTENT (IN) for <code>vdexpm1</code> , <code>vmsexpm1</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Expm1 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \text{Ln}(\text{FLT_MAX})$
double precision	$a[i] < \text{Ln}(\text{DBL_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdexpm1</code> , <code>vmsexpm1</code> REAL, INTENT (OUT) for <code>vsexpm1</code> , <code>vmsexpm1</code>	Array that specifies the output vector <i>y</i> .

Name	Type	Description
------	------	-------------

	DOUBLE PRECISION, INTENT (OUT) for vdexpm1, vmdexpm1	
--	---	--

Description

The `v?Expm1` function computes an exponential of vector elements decreased by 1.

Special Values for Real Function `v?Expm1(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	+0		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	-1		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Ln`

Computes natural logarithm of vector elements.

Syntax

```
call vsln( n, a, y )
call vmsln( n, a, y, mode )
call vdln( n, a, y )
call vmdln( n, a, y, mode )
call vcln( n, a, y )
call vmcln( n, a, y, mode )
call vzln( n, a, y )
call vmzln( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdln</code> , <code>vmdln</code> COMPLEX for <code>vcln</code> , <code>vmcln</code> DOUBLE COMPLEX for <code>vzln</code> , <code>vmzln</code> REAL, INTENT (IN) for <code>vsln</code> , <code>vmsln</code> DOUBLE PRECISION, INTENT (IN) for <code>vdln</code> , <code>vmdln</code>	Array that specifies the input vector <code>a</code> .

Name	Type	Description
	COMPLEX, INTENT (IN) for <code>vcln</code> , <code>vmcln</code>	
	DOUBLE COMPLEX, INTENT (IN) for <code>vzln</code> , <code>vmzln</code>	
<code>mode</code>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	DOUBLE PRECISION for <code>vdln</code> , <code>vmdl</code> COMPLEX for <code>vcln</code> , <code>vmcln</code> DOUBLE COMPLEX for <code>vzln</code> , <code>vmzln</code> REAL, INTENT (OUT) for <code>vsln</code> , <code>vmsln</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdln</code> , <code>vmdl</code> COMPLEX, INTENT (OUT) for <code>vcln</code> , <code>vmcln</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzln</code> , <code>vmzln</code>	Array that specifies the output vector <code>y</code> .

Description

The `v?Ln` function computes natural logarithm of vector elements.

Special Values for Real Function `v?Ln(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Ln(z)`

$\text{RE}(z)$ $i \cdot \text{IM}(z)$	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i \cdot \infty$	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$			

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN INVALID
+i·0	+∞+i·π		-∞+i·π ZERODIVID E	-∞+i·0 ZERODIVID E		+∞+i·0	QNAN+i·QNAN INVALID
-i·0	+∞-i·π		-∞-i·π ZERODIVID E	-∞-i·0 ZERODIVID E		+∞-i·0	QNAN+i·QNAN INVALID
-i·Y	+∞-i·π					+∞-i·0	QNAN+i·QNAN INVALID
-i·∞	+∞ - i · $\frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN	QNAN+i·QNAN INVALID

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN

v?Log10

Computes denary logarithm of vector elements.

Syntax

```
call vslog10( n, a, y )
call vmslog10( n, a, y, mode )
call vdlog10( n, a, y )
call vmdlog10( n, a, y, mode )
call vclog10( n, a, y )
call vmclog10( n, a, y, mode )
call vzlog10( n, a, y )
call vmzlog10( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdlog10, vmdlog10	Array that specifies the input vector <i>a</i> .

Name	Type	Description
	COMPLEX for <code>vclog10</code> , <code>vmclog10</code>	
	DOUBLE COMPLEX for <code>vzlog10</code> , <code>vmzlog10</code>	
	REAL, INTENT(IN) for <code>vslog10</code> , <code>vmslog10</code>	
	DOUBLE PRECISION, INTENT(IN) for <code>vdlog10</code> , <code>vmdlog10</code>	
	COMPLEX, INTENT(IN) for <code>vclog10</code> , <code>vmclog10</code>	
	DOUBLE COMPLEX, INTENT(IN) for <code>vzlog10</code> , <code>vmzlog10</code>	
<code>mode</code>	INTEGER (KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	DOUBLE PRECISION for <code>vdlog10</code> , <code>vmdlog10</code>	Array that specifies the output vector <code>y</code> .
	COMPLEX for <code>vclog10</code> , <code>vmclog10</code>	
	DOUBLE COMPLEX for <code>vzlog10</code> , <code>vmzlog10</code>	
	REAL, INTENT(OUT) for <code>vslog10</code> , <code>vmslog10</code>	
	DOUBLE PRECISION, INTENT(OUT) for <code>vdlog10</code> , <code>vmdlog10</code>	
	COMPLEX, INTENT(OUT) for <code>vclog10</code> , <code>vmclog10</code>	
	DOUBLE COMPLEX, INTENT(OUT) for <code>vzlog10</code> , <code>vmzlog10</code>	

Description

The `v?Log10` function computes a denary logarithm of vector elements.

Special Values for Real Function `v?Log10(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		

Argument	Result	VM Error Status	Exception
SNAN	QNaN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Log10(z)$

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{4} \frac{1}{\ln(10)}$	$+\infty+i$ QNaN INVALID
$+i\cdot Y$	$+\infty + i \frac{\pi}{\ln(10)}$					$+\infty+i\cdot 0$	QNaN+i·QNaN INVALID
$+i\cdot 0$	$+\infty + i \frac{\pi}{\ln(10)}$		$-\infty + i \frac{\pi}{\ln(10)}$ ZERODRIVE	$-\infty+i\cdot 0$ ZERODRIVE		$+\infty+i\cdot 0$	QNaN+i·QNaN INVALID
$-i\cdot 0$	$+\infty - i \frac{\pi}{\ln(10)}$		$-\infty - i \frac{\pi}{\ln(10)}$ ZERODIVID E	$-\infty-i\cdot 0$ ZERODIVID E		$+\infty-i\cdot 0$	QNaN-i·QNaN INVALID
$-i\cdot Y$	$+\infty - i \frac{\pi}{\ln(10)}$					$+\infty-i\cdot 0$	QNaN+i·QNaN INVALID
$-i\cdot\infty$	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{4} \frac{1}{\ln(10)}$	$+\infty+i$ QNaN
$+i\cdot$ NAN	$+\infty+i$ QNaN	QNaN+i·QNaN INVALID	QNaN+i·QNaN INVALID	QNaN+i·QNaN INVALID	QNaN+i·QNaN INVALID	$+\infty+i$ QNaN	QNaN+i·QNaN INVALID

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN

$v?Log1p$

Computes a natural logarithm of vector elements that are increased by 1.

Syntax

```
call vslog1p( n, a, y )
call vmslog1p( n, a, y, mode )
call vdlog1p( n, a, y )
call vmdlog1p( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	DOUBLE PRECISION for <code>vdlog1p</code> , <code>vmdlog1p</code> REAL, INTENT(IN) for <code>vslog1p</code> , <code>vmslog1p</code> DOUBLE PRECISION, INTENT(IN) for <code>vdlog1p</code> , <code>vmdlog1p</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdlog1p</code> , <code>vmdlog1p</code> REAL, INTENT(OUT) for <code>vslog1p</code> , <code>vmslog1p</code> DOUBLE PRECISION, INTENT(OUT) for <code>vdlog1p</code> , <code>vmdlog1p</code>	Array that specifies the output vector <i>y</i> .

Description

The `v?Log1p` function computes a natural logarithm of vector elements that are increased by 1.

Special Values for Real Function `v?Log1p(x)`

Argument	Result	VM Error Status	Exception
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -1$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

Trigonometric Functions

`v?Cos`

Computes cosine of vector elements.

Syntax

```
call vscol( n, a, y )
call vmcos( n, a, y, mode )
call vdcos( n, a, y )
call vmdcos( n, a, y, mode )
```

```
call vccos( n, a, y )
call vmccos( n, a, y, mode )
call vzcoss( n, a, y )
call vmzcoss( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdcoss</code> , <code>vmdcoss</code> COMPLEX for <code>vccoss</code> , <code>vmccoss</code> DOUBLE PRECISION for <code>vzcoss</code> , <code>vmzcoss</code> REAL, INTENT (IN) for <code>vscoss</code> , <code>vmscoss</code> DOUBLE PRECISION, INTENT (IN) for <code>vdcoss</code> , <code>vmdcoss</code> COMPLEX, INTENT (IN) for <code>vccoss</code> , <code>vmccoss</code> DOUBLE PRECISION, INTENT (IN) for <code>vzcoss</code> , <code>vmzcoss</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdcoss</code> , <code>vmdcoss</code> COMPLEX for <code>vccoss</code> , <code>vmccoss</code> DOUBLE PRECISION for <code>vzcoss</code> , <code>vmzcoss</code> REAL, INTENT (OUT) for <code>vscoss</code> , <code>vmscoss</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdcoss</code> , <code>vmdcoss</code> COMPLEX, INTENT (OUT) for <code>vccoss</code> , <code>vmccoss</code>	Array that specifies the output vector <i>y</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT (OUT) for vzcos, vmzcos	

Description

The `v?Cos` function computes cosine of vector elements.

Note that arguments $abs(a[i]) \leq 2^{13}$ and $abs(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Cos(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Cos}(z) = \text{Cosh}(i*z).$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

v?Sin

Computes sine of vector elements.

Syntax

```
call vssin( n, a, y )
call vmssin( n, a, y, mode )
call vdsin( n, a, y )
call vmdsin( n, a, y, mode )
call vcsin( n, a, y )
call vmcsin( n, a, y, mode )
call vzsine( n, a, y )
call vmzsin( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <i>vdsin</i> , <i>vmdsin</i> COMPLEX for <i>vcsin</i> , <i>vmcsin</i> DOUBLE PRECISION for <i>vzsin</i> , <i>vmzsin</i> REAL, INTENT (IN) for <i>vssin</i> , <i>vmssin</i> DOUBLE PRECISION, INTENT (IN) for <i>vdsin</i> , <i>vmdsin</i> COMPLEX, INTENT (IN) for <i>vcsin</i> , <i>vmcsin</i> DOUBLE PRECISION, INTENT (IN) for <i>vzsin</i> , <i>vmzsin</i>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vdsin</i> , <i>vmdsin</i> COMPLEX for <i>vcsin</i> , <i>vmcsin</i> DOUBLE PRECISION for <i>vzsin</i> , <i>vmzsin</i> REAL, INTENT (OUT) for <i>vssin</i> , <i>vmssin</i> DOUBLE PRECISION, INTENT (OUT) for <i>vdsin</i> , <i>vmdsin</i> COMPLEX, INTENT (OUT) for <i>vcsin</i> , <i>vmcsin</i> DOUBLE PRECISION, INTENT (OUT) for <i>vzsin</i> , <i>vmzsin</i>	Array that specifies the output vector <i>y</i> .

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Sin(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

v?SinCos

Computes sine and cosine of vector elements.

Syntax

```
call vssincos( n, a, y, z )
call vmssincos( n, a, y, z, mode )
call vdsincos( n, a, y, z )
call vmdsincos( n, a, y, z, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdsincos, vmdsincos	Array that specifies the input vector <i>a</i> .

Name	Type	Description
	REAL, INTENT(IN) for <code>vssincos</code> , <code>vmssincos</code>	
	DOUBLE PRECISION, INTENT(IN) for <code>vdsincos</code> , <code>vmdsincos</code>	
<code>mode</code>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y, z</code>	DOUBLE PRECISION for <code>vdsincos</code> , <code>vmdsincos</code>	Arrays that specify the output vectors y (for sine values) and z (for cosine values).
	REAL, INTENT(OUT) for <code>vssincos</code> , <code>vmssincos</code>	
	DOUBLE PRECISION, INTENT(OUT) for <code>vdsincos</code> , <code>vmdsincos</code>	

Description

The function computes sine and cosine of vector elements.

Note that arguments $abs(a[i]) \leq 2^{13}$ and $abs(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function `v?SinCos(x)`

Argument	Result 1	Result 2	VM Error Status	Exception
+0	+0	+1		
-0	-0	+1		
$+\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN	QNAN		
SNAN	QNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

v?CIS

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

```
call vccis( n, a, y )
call vmccis( n, a, y, mode )
call vzcis( n, a, y )
call vmzcis( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vzcis, vmzcis REAL, INTENT (IN) for vccis, vmccis DOUBLE PRECISION, INTENT (IN) for vzcis, vmzcis	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE COMPLEX for vzcis, vmzcis COMPLEX, INTENT (OUT) for vccis, vmccis DOUBLE COMPLEX, INTENT (OUT) for vzcis, vmzcis	Array that specifies the output vector <i>y</i> .

Description

The v?CIS function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?CIS(x)$

x	CIS(x)
+ ∞	QNAN+i·QNAN INVALID
+ 0	+1+i·0
- 0	+1-i·0
- ∞	QNAN+i·QNAN INVALID
NAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the argument is `SNAN`
- raises `INVALID` exception and sets the VM Error Status to `VML_STATUS_ERRDOM` for $x=+\infty$, $x=-\infty$

 $v?Tan$

Computes tangent of vector elements.

Syntax

```
call vstan( n, a, y )
call vmstan( n, a, y, mode )
call vdtan( n, a, y )
call vmdtan( n, a, y, mode )
call vctan( n, a, y )
call vmctan( n, a, y, mode )
call vztan( n, a, y )
call vmztan( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdtan</code> , <code>vmdtan</code> COMPLEX for <code>vctan</code> , <code>vmctan</code> DOUBLE COMPLEX for <code>vztan</code> , <code>vmztan</code> REAL, INTENT (IN) for <code>vstan</code> , <code>vmstan</code>	Array that specifies the input vector <code>a</code> .

Name	Type	Description
	DOUBLE PRECISION, INTENT (IN) for vdtan, vmdtan	
	COMPLEX, INTENT (IN) for vctan, vmctan	
	DOUBLE COMPLEX, INTENT (IN) for vztan, vmztan	
mode	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdtan, vmdtan	Array that specifies the output vector <i>y</i> .
	COMPLEX for vctan, vmctan	
	DOUBLE COMPLEX for vztan, vmztan	
	REAL, INTENT (OUT) for vstan, vmstan	
	DOUBLE PRECISION, INTENT (OUT) for vdtan, vmdtan	
	COMPLEX, INTENT (OUT) for vctan, vmctan	
	DOUBLE COMPLEX, INTENT (OUT) for vztan, vmztan	

Description

The $v?Tan$ function computes tangent of vector elements.

Note that arguments $abs(a[i]) \leq 2^{13}$ and $abs(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function $v?Tan(x)$

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\tan(z) = -i * \tanh(i * z).$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

v?Acos

Computes inverse cosine of vector elements.

Syntax

```
call vsacos( n, a, y )
call vmsacos( n, a, y, mode )
call vdacos( n, a, y )
call vmdacos( n, a, y, mode )
call vcacos( n, a, y )
call vmcacos( n, a, y, mode )
call vzasacos( n, a, y )
call vmzasacos( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <i>vdacos</i> , <i>vmdacos</i> COMPLEX for <i>vcacos</i> , <i>vmcacos</i> DOUBLE COMPLEX for <i>vzasacos</i> , <i>vmzasacos</i> REAL, INTENT (IN) for <i>vsacos</i> , <i>vmsacos</i> DOUBLE PRECISION, INTENT (IN) for <i>vdacos</i> , <i>vmdacos</i> COMPLEX, INTENT (IN) for <i>vcacos</i> , <i>vmcacos</i>	Array that specifies the input vector <i>a</i> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT (IN) for <code>vzacos</code> , <code>vmzacos</code>	
<code>mode</code>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	DOUBLE PRECISION for <code>vdacos</code> , <code>vmdacos</code> COMPLEX for <code>vcacos</code> , <code>vmcacos</code> DOUBLE COMPLEX for <code>vzacos</code> , <code>vmzacos</code> REAL, INTENT (OUT) for <code>vsacos</code> , <code>vmsacos</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdacos</code> , <code>vmdacos</code> COMPLEX, INTENT (OUT) for <code>vcacos</code> , <code>vmcacos</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzacos</code> , <code>vmzacos</code>	Array that specifies the output vector <code>y</code> .

Description

The `v?Acos` function computes inverse cosine of vector elements.

Special Values for Real Function `v?Acos(x)`

Argument	Result	VM Error Status	Exception
+0	$+\pi/2$		
-0	$+\pi/2$		
+1	+0		
-1	$+\pi$		
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Acos(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+\frac{3\pi}{4} - i \cdot \infty$	$+\frac{\pi}{2} - i \cdot \infty$	$+\frac{\pi}{4} - i \cdot \infty$	QNAN- $i\cdot\infty$			

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·Y	$+\pi-i\cdot\infty$					$+0-i\cdot\infty$	QNAN+i·QNAN
+i·0	$+\pi-i\cdot\infty$		$+\frac{\pi}{2}-i\cdot 0$	$+\frac{\pi}{2}-i\cdot 0$		$+0-i\cdot\infty$	QNAN+i·QNAN
-i·0	$+\pi+i\cdot\infty$		$+\frac{\pi}{2}+i\cdot\infty$	$+\frac{\pi}{2}+i\cdot\infty$		$+0+i\cdot\infty$	QNAN+i·QNAN
-i·Y	$+\pi+i\cdot\infty$					$+0+i\cdot\infty$	QNAN+i·QNAN
-i·∞	$+\frac{3\pi}{4}+i\cdot\infty$	$+\frac{\pi}{2}+i\cdot\infty$	$+\frac{\pi}{2}+i\cdot\infty$	$+\frac{\pi}{2}+i\cdot\infty$	$+\frac{\pi}{2}+i\cdot\infty$	$+\frac{\pi}{4}+i\cdot\infty$	QNAN+i·∞
+i·NAN	QNAN+i·∞	QNAN+i·QNAN	$+\frac{\pi}{2}+i\cdot\text{QNAN}$	$+\frac{\pi}{2}+i\cdot\text{QNAN}$	QNAN+i·QNAN	QNAN+i·∞	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $\text{Acos}(\text{CONJ}(z)) = \text{CONJ}(\text{Acos}(z))$.

v?Asin

Computes inverse sine of vector elements.

Syntax

```
call vsasin( n, a, y )
call vmsasin( n, a, y, mode )
call vdasin( n, a, y )
call vmdasin( n, a, y, mode )
call vcasin( n, a, y )
call vmcasin( n, a, y, mode )
call vzasin( n, a, y )
call vmzasin( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdasin</code> , <code>vmdasin</code> COMPLEX for <code>vcasin</code> , <code>vmcasin</code> DOUBLE COMPLEX for <code>vzasin</code> , <code>vmzasin</code>	Array that specifies the input vector <code>a</code> .

Name	Type	Description
	REAL, INTENT (IN) for vsasin, vmsasin	
	DOUBLE PRECISION, INTENT (IN) for vdasin, vmdasin	
	COMPLEX, INTENT (IN) for vcasin, vmcasin	
	DOUBLE COMPLEX, INTENT (IN) for vzasin, vmzasin	
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdasin, vmdasin COMPLEX for vcasin, vmcasin DOUBLE COMPLEX for vzasin, vmzasin REAL, INTENT (OUT) for vsasin, vmsasin DOUBLE PRECISION, INTENT (OUT) for vdasin, vmdasin COMPLEX, INTENT (OUT) for vcasin, vmcasin DOUBLE COMPLEX, INTENT (OUT) for vzasin, vmzasin	Array that specifies the output vector <i>y</i> .

Description

The `v?Asin` function computes inverse sine of vector elements.

Special Values for Real Function `v?Asin(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	$+\pi/2$		
-1	$-\pi/2$		
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$\text{Asin}(z) = -i \cdot \text{Asinh}(i \cdot z)$.

v?Atan

Computes inverse tangent of vector elements.

Syntax

```
call vsatan( n, a, y )
call vmsatan( n, a, y, mode )
call vdatan( n, a, y )
call vmdatan( n, a, y, mode )
call vcatan( n, a, y )
call vmcatan( n, a, y, mode )
call vzatan( n, a, y )
call vmzatan( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdatan, vmdatan COMPLEX for vcatan, vmcatan DOUBLE COMPLEX for vzatan, vmzatan REAL, INTENT (IN) for vsatan, vmsatan DOUBLE PRECISION, INTENT (IN) for vdatan, vmdatan COMPLEX, INTENT (IN) for vcatan, vmcatan DOUBLE COMPLEX, INTENT (IN) for vzatan, vmzatan	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdatan</code> , <code>vmdatan</code> COMPLEX for <code>vcatan</code> , <code>vmcatan</code> DOUBLE COMPLEX for <code>vzatan</code> , <code>vmzatan</code> REAL, INTENT (OUT) for <code>vsatan</code> , <code>vmsatan</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdatan</code> , <code>vmdatan</code> COMPLEX, INTENT (OUT) for <code>vcatan</code> , <code>vmcatan</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzatan</code> , <code>vmzatan</code>	Array that specifies the output vector y .

Description

The `v?Atan` function computes inverse tangent of vector elements.

Special Values for Real Function `v?Atan(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	$+\pi/2$		
$-\infty$	$-\pi/2$		
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Atan}(z) = -i \cdot \text{Atanh}(i \cdot z).$$

`v?Atan2`

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
call vsatan2( n, a, b, y )
call vmsatan2( n, a, b, y, mode )
call vdatan2( n, a, b, y )
call vmdatan2( n, a, b, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a, b	DOUBLE PRECISION for <code>vdatan2</code> , <code>vmdatan2</code> REAL, INTENT (IN) for <code>vsatan2</code> , <code>vmsatan2</code> DOUBLE PRECISION, INTENT (IN) for <code>vdatan2</code> , <code>vmdatan2</code>	Arrays that specify the input vectors a and b .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vm1SetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdatan2</code> , <code>vmdatan2</code> REAL, INTENT (OUT) for <code>vsatan2</code> , <code>vmsatan2</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdatan2</code> , <code>vmdatan2</code>	Array that specifies the output vector y .

Description

The `v?Atan2` function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector y are computed as the four-quadrant arctangent of $a[i] / b[i]$.

Special values for Real Function `v?Atan2(x)`

Argument 1	Argument 2	Result	Exception
$-\infty$	$-\infty$	$-3*\pi/4$	
$-\infty$	$X < +0$	$-\pi/2$	
$-\infty$	-0	$-\pi/2$	
$-\infty$	$+0$	$-\pi/2$	
$-\infty$	$X > +0$	$-\pi/2$	
$-\infty$	$+\infty$	$-\pi/4$	
$X < +0$	$-\infty$	$-\pi$	
$X < +0$	-0	$-\pi/2$	
$X < +0$	$+0$	$-\pi/2$	
$X < +0$	$+\infty$	-0	
-0	$-\infty$	$-\pi$	
-0	$X < +0$	$-\pi$	
-0	-0	$-\pi$	
-0	$+0$	-0	
-0	$X > +0$	-0	
-0	$+\infty$	-0	

Argument 1	Argument 2	Result	Exception
+0	$-\infty$	$+\pi$	
+0	$X < +0$	$+\pi$	
+0	-0	$+\pi$	
+0	+0	+0	
+0	$X > +0$	+0	
+0	$+\infty$	+0	
$X > +0$	$-\infty$	$+\pi$	
$X > +0$	-0	$+\pi/2$	
$X > +0$	+0	$+\pi/2$	
$X > +0$	$+\infty$	+0	
$+\infty$	$-\infty$	$-3*\pi/4$	
$+\infty$	$X < +0$	$+\pi/2$	
$+\infty$	-0	$+\pi/2$	
$+\infty$	+0	$+\pi/2$	
$+\infty$	$X > +0$	$+\pi/2$	
$+\infty$	$+\infty$	$+\pi/4$	
$X > +0$	QNAN	QNAN	
$X > +0$	SNAN	QNAN	INVALID
QNAN	$X > +0$	QNAN	
SNAN	$X > +0$	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

Hyperbolic Functions

v?Cosh

Computes hyperbolic cosine of vector elements.

Syntax

```
call vscosh( n, a, y )
call vmscosh( n, a, y, mode )
call vdcosh( n, a, y )
call vmdcosh( n, a, y, mode )
call vccosh( n, a, y )
call vmccosh( n, a, y, mode )
call vzcosh( n, a, y )
call vmzcosh( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <i>vdcosh</i> , <i>vmdcosh</i> COMPLEX for <i>vccosh</i> , <i>vmccosh</i> DOUBLE COMPLEX for <i>vzcosh</i> , <i>vmzcosh</i> REAL, INTENT (IN) for <i>vscosh</i> , <i>vmscosh</i> DOUBLE PRECISION, INTENT (IN) for <i>vdcosh</i> , <i>vmdcosh</i> COMPLEX, INTENT (IN) for <i>vccosh</i> , <i>vmccosh</i> DOUBLE COMPLEX, INTENT (IN) for <i>vzcosh</i> , <i>vmzcosh</i>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Precision Overflow Thresholds for Real *v?Cosh* Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT_MAX}) - \ln 2 < a[i] < \ln(\text{FLT_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL_MAX}) - \ln 2 < a[i] < \ln(\text{DBL_MAX}) + \ln 2$

Precision overflow thresholds for the complex *v?Cosh* function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vdcosh</i> , <i>vmdcosh</i> COMPLEX for <i>vccosh</i> , <i>vmccosh</i> DOUBLE COMPLEX for <i>vzcosh</i> , <i>vmzcosh</i> REAL, INTENT (OUT) for <i>vscosh</i> , <i>vmscosh</i> DOUBLE PRECISION, INTENT (OUT) for <i>vdcosh</i> , <i>vmdcosh</i> COMPLEX, INTENT (OUT) for <i>vccosh</i> , <i>vmccosh</i> DOUBLE COMPLEX, INTENT (OUT) for <i>vzcosh</i> , <i>vmzcosh</i>	Array that specifies the output vector <i>y</i> .

Description

The `v?Cosh` function computes hyperbolic cosine of vector elements.

Special Values for Real Function `v?Cosh(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Cosh(z)`

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN-i·0 INVALID	QNAN+i·0 INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	+∞-i·0		+1-i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+∞+i·0		+1+i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	+∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·0 INVALID	QNAN-i·0 INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN-i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when $RE(z)$, $IM(z)$ are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $Cosh(CONJ(z)) = CONJ(Cosh(z))$
- $Cosh(-z) = Cosh(z)$.

`v?Sinh`

Computes hyperbolic sine of vector elements.

Syntax

```
call vssinh( n, a, y )
```

```
call vmssinh( n, a, y, mode )
```

```
call vdsinh( n, a, y )
```

```
call vmdsinh( n, a, y, mode )
```

```
call vcsinh( n, a, y )
call vmcsinh( n, a, y, mode )
call vzsinh( n, a, y )
call vmzsinh( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh REAL, INTENT (IN) for vssinh, vmssinh DOUBLE PRECISION, INTENT (IN) for vdsinh, vmdsinh COMPLEX, INTENT (IN) for vcsinh, vmcsinh DOUBLE COMPLEX, INTENT (IN) for vzsinh, vmzsinh	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Sinh Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT_MAX}) - \ln 2 < a[i] < \ln(\text{FLT_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL_MAX}) - \ln 2 < a[i] < \ln(\text{DBL_MAX}) + \ln 2$

Precision overflow thresholds for the complex v?Sinh function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh	Array that specifies the output vector <i>y</i> .

Name	Type	Description
	REAL, INTENT (OUT) for <code>vssinh</code> , <code>vmssinh</code>	
	DOUBLE PRECISION, INTENT (OUT) for <code>vdsinh</code> , <code>vmdsinh</code>	
	COMPLEX, INTENT (OUT) for <code>vcsinh</code> , <code>vmcsinh</code>	
	DOUBLE COMPLEX, INTENT (OUT) for <code>vzsinh</code> , <code>vmzsinh</code>	

Description

The `v?Sinh` function computes hyperbolic sine of vector elements.

Special Values for Real Function `v?Sinh(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
X > overflow	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	$-\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$+\infty$	$+\infty$		
$-\infty$	$-\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Sinh(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$-\infty+i\cdot\text{QNAN}$ INVALID	QNAN+i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i\cdot\text{QNAN}$ INVALID	QNAN+i·QNAN
$+i\cdot Y$	$-\infty\cdot\text{Cos}(Y)+$ $i\cdot\infty\cdot\text{Sin}(Y)$					$+\infty\cdot\text{CIS}(Y)$	QNAN+i·QNAN
$+i\cdot 0$	$-\infty+i\cdot 0$		-0+i·0	+0+i·0		$+\infty+i\cdot 0$	QNAN+i·0
$-i\cdot 0$	$-\infty-i\cdot 0$		-0-i·0	+0-i·0		$+\infty-i\cdot 0$	QNAN-i·0
$-i\cdot Y$	$-\infty\cdot\text{Cos}(Y)+$ $i\cdot\infty\cdot\text{Sin}(Y)$					$+\infty\cdot\text{CIS}(Y)$	QNAN+i·QNAN
$-i\cdot\infty$	$-\infty+i\cdot\text{QNAN}$ INVALID	QNAN+i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i\cdot\text{QNAN}$ INVALID	QNAN+i·QNAN
$+i\cdot\text{NAN}$	$-\infty+i\cdot\text{QNAN}$	QNAN+i·QNAN	-0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	$+\infty+i\cdot\text{QNAN}$	QNAN+i·QNAN

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when `RE(z)`, `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

- $\text{Sinh}(\text{CONJ}(z)) = \text{CONJ}(\text{Sinh}(z))$
- $\text{Sinh}(-z) = -\text{Sinh}(z)$.

v?Tanh

Computes hyperbolic tangent of vector elements.

Syntax

```
call vstanh( n, a, y )
call vmstanh( n, a, y, mode )
call vdtanh( n, a, y )
call vmdtanh( n, a, y, mode )
call vctanh( n, a, y )
call vmctanh( n, a, y, mode )
call vztanh( n, a, y )
call vmztanh( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdtanh, vmdtanh COMPLEX for vctanh, vmctanh DOUBLE COMPLEX for vztanh, vmztanh REAL, INTENT (IN) for vstanh, vmstanh DOUBLE PRECISION, INTENT (IN) for vdtanh, vmdtanh COMPLEX, INTENT (IN) for vctanh, vmctanh DOUBLE COMPLEX, INTENT (IN) for vztanh, vmztanh	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdtanh</code> , <code>vmdtanh</code> COMPLEX for <code>vctanh</code> , <code>vmctanh</code> DOUBLE COMPLEX for <code>vztanh</code> , <code>vmztanh</code> REAL, INTENT (OUT) for <code>vstanh</code> , <code>vmstanh</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdtanh</code> , <code>vmdtanh</code> COMPLEX, INTENT (OUT) for <code>vctanh</code> , <code>vmctanh</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vztanh</code> , <code>vmztanh</code>	Array that specifies the output vector y .

Description

The `v?Tanh` function computes hyperbolic tangent of vector elements.

Special Values for Real Function `v?Tanh(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	+1	
$-\infty$	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Tanh(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	-1+i·0	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+1+i·0	QNAN+i·QNAN
$+i\cdot Y$	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN+i·QNAN
$+i\cdot 0$	-1+i·0		-0+i·0	+0+i·0		+1+i·0	QNAN+i·0
$-i\cdot 0$	-1-i·0		-0-i·0	+0-i·0		+1-i·0	QNAN-i·0
$-i\cdot Y$	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN+i·QNAN
$-i\cdot\infty$	-1-i·0	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+1-i·0	QNAN+i·QNAN
$+i\cdot\text{NAN}$	-1+i·0	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+1+i·0	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $\text{Tanh}(\text{CONJ}(z)) = \text{CONJ}(\text{Tanh}(z))$
- $\text{Tanh}(-z) = -\text{Tanh}(z)$.

v?Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

```
call vsacosh( n, a, y )
call vmsacosh( n, a, y, mode )
call vdacosh( n, a, y )
call vmdacosh( n, a, y, mode )
call vcacosh( n, a, y )
call vmcacosh( n, a, y, mode )
call vzacosh( n, a, y )
call vmzacosh( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdacosh</code> , <code>vmdacosh</code> COMPLEX for <code>vcacosh</code> , <code>vmcacosh</code> DOUBLE COMPLEX for <code>vzacosh</code> , <code>vmzacosh</code> REAL, INTENT(IN) for <code>vsacosh</code> , <code>vmsacosh</code> DOUBLE PRECISION, INTENT(IN) for <code>vdacosh</code> , <code>vmdacosh</code> COMPLEX, INTENT(IN) for <code>vcacosh</code> , <code>vmcacosh</code> DOUBLE COMPLEX, INTENT(IN) for <code>vzacosh</code> , <code>vmzacosh</code>	Array that specifies the input vector <code>a</code> .
<code>mode</code>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdacosh</code> , <code>vmdacosh</code> COMPLEX for <code>vcacosh</code> , <code>vmcacosh</code> DOUBLE COMPLEX for <code>vzacosh</code> , <code>vmzacosh</code> REAL, INTENT (OUT) for <code>vsacosh</code> , <code>vmsacosh</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdacosh</code> , <code>vmdacosh</code> COMPLEX, INTENT (OUT) for <code>vcacosh</code> , <code>vmcacosh</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzacosh</code> , <code>vmzacosh</code>	Array that specifies the output vector y .

Description

The $v?Acosh$ function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Special Values for Real Function $v?Acosh(x)$

Argument	Result	VM Error Status	Exception
+1	+0		
$X < +1$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Acosh(z)$

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/4$	$+\infty+i\cdot\text{QNAN}$
$+i\cdot Y$	$+\infty+i\cdot\pi$					$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot 0$	$+\infty+i\cdot\pi$		$+0+i\cdot\pi/2$	$+0+i\cdot\pi/2$		$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot 0$	$+\infty+i\cdot\pi$		$+0+i\cdot\pi/2$	$+0+i\cdot\pi/2$		$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot Y$	$+\infty+i\cdot\pi$					$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot\infty$	$+\infty - i \cdot \frac{3\pi}{4}$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/4$	$+\infty+i\cdot\text{QNAN}$
$+i\cdot\text{NAN}$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- `Acosh(CONJ(z))=CONJ(Acosh(z))`.

v?Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax

```
call vsasinh( n, a, y )
call vmsasinh( n, a, y, mode )
call vdasinh( n, a, y )
call vmdasinh( n, a, y, mode )
call vcasinh( n, a, y )
call vmcasinh( n, a, y, mode )
call vzasinh( n, a, y )
call vmzasinh( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdasinh</code> , <code>vmdasinh</code> COMPLEX for <code>vcasinh</code> , <code>vmcasinh</code> DOUBLE COMPLEX for <code>vzasinh</code> , <code>vmzasinh</code> REAL, INTENT(IN) for <code>vsasinh</code> , <code>vmsasinh</code> DOUBLE PRECISION, INTENT(IN) for <code>vdasinh</code> , <code>vmdasinh</code> COMPLEX, INTENT(IN) for <code>vcasinh</code> , <code>vmcasinh</code> DOUBLE COMPLEX, INTENT(IN) for <code>vzasinh</code> , <code>vmzasinh</code>	Array that specifies the input vector <code>a</code> .
<code>mode</code>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdasinh</code> , <code>vmdasinh</code> COMPLEX for <code>vcasinh</code> , <code>vmcasinh</code> DOUBLE COMPLEX for <code>vzasinh</code> , <code>vmzasinh</code> REAL, INTENT (OUT) for <code>vsasinh</code> , <code>vmsasinh</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdasinh</code> , <code>vmdasinh</code> COMPLEX, INTENT (OUT) for <code>vcasinh</code> , <code>vmcasinh</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzasinh</code> , <code>vmzasinh</code>	Array that specifies the output vector y .

Description

The `v?Asinh` function computes inverse hyperbolic sine of vector elements.

Special Values for Real Function `v?Asinh(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Asinh(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$-\infty+i\cdot\pi/4$	$-\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/2$	$+\infty+i\cdot\pi/4$	$+\infty+i\cdot\text{QNAN}$
$+i\cdot Y$	$-\infty+i\cdot 0$					$+\infty+i\cdot 0$	QNAN+i·QNAN
$+i\cdot 0$	$+\infty+i\cdot 0$		$+0+i\cdot 0$	$+0+i\cdot 0$		$+\infty+i\cdot 0$	QNAN+i·QNAN
$-i\cdot 0$	$-\infty-i\cdot 0$		$-0-i\cdot 0$	$+0-i\cdot 0$		$+\infty-i\cdot 0$	QNAN-i·QNAN
$-i\cdot Y$	$-\infty-i\cdot 0$					$+\infty-i\cdot 0$	QNAN+i·QNAN
$-i\cdot\infty$	$-\infty-i\cdot\pi/4$	$-\infty-i\cdot\pi/2$	$-\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/2$	$+\infty-i\cdot\pi/4$	$+\infty+i\cdot\text{QNAN}$
$+i\cdot\text{NAN}$	$-\infty+i\cdot\text{QNAN}$	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	$+\infty+i\cdot\text{QNAN}$	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`

- $\text{Asinh}(\text{CONJ}(z)) = \text{CONJ}(\text{Asinh}(z))$
- $\text{Asinh}(-z) = -\text{Asinh}(z)$.

v?Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax

```
call vsatanh( n, a, y )
call vmsatanh( n, a, y, mode )
call vdatanh( n, a, y )
call vmdatanh( n, a, y, mode )
call vcatanh( n, a, y )
call vmcatanh( n, a, y, mode )
call vzatanh( n, a, y )
call vmzatanh( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <i>vdatanh</i> , <i>vmdatanh</i> COMPLEX for <i>vcatanh</i> , <i>vmcatanh</i> DOUBLE COMPLEX for <i>vzatanh</i> , <i>vmzatanh</i> REAL, INTENT(IN) for <i>vsatanh</i> , <i>vmsatanh</i> DOUBLE PRECISION, INTENT(IN) for <i>vdatanh</i> , <i>vmdatanh</i> COMPLEX, INTENT(IN) for <i>vcatanh</i> , <i>vmcatanh</i> DOUBLE COMPLEX, INTENT(IN) for <i>vzatanh</i> , <i>vmzatanh</i>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdatanh</code> , <code>vmdatanh</code> COMPLEX for <code>vcatanh</code> , <code>vmcatanh</code> DOUBLE COMPLEX for <code>vzatanh</code> , <code>vmzatanh</code> REAL, INTENT (OUT) for <code>vsatanh</code> , <code>vmsatanh</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdatanh</code> , <code>vmdatanh</code> COMPLEX, INTENT (OUT) for <code>vcatanh</code> , <code>vmcatanh</code> DOUBLE COMPLEX, INTENT (OUT) for <code>vzatanh</code> , <code>vmzatanh</code>	Array that specifies the output vector y .

Description

The `v?Atanh` function computes inverse hyperbolic tangent of vector elements.

Special Values for Real Function `v?Atanh(x)`

Argument	Result	VM Error Status	Exception
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?Atanh(z)`

RE(z) i·IM(z))	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$-0+i\cdot\pi/2$	$-0+i\cdot\pi/2$	$-0+i\cdot\pi/2$	$+0+i\cdot\pi/2$	$+0+i\cdot\pi/2$	$+0+i\cdot\pi/2$	$+0+i\cdot\pi/2$
$+i\cdot Y$	$-0+i\cdot\pi/2$					$+0+i\cdot\pi/2$	QNAN+i·QNAN
$+i\cdot 0$	$-0+i\cdot\pi/2$		$-0+i\cdot 0$	$+0+i\cdot 0$		$+0+i\cdot\pi/2$	QNAN+i·QNAN
$-i\cdot 0$	$-0-i\cdot\pi/2$		$-0-i\cdot 0$	$+0-i\cdot 0$		$+0-i\cdot\pi/2$	QNAN-i·QNAN
$-i\cdot Y$	$-0-i\cdot\pi/2$					$+0-i\cdot\pi/2$	QNAN+i·QNAN
$-i\cdot\infty$	$-0-i\cdot\pi/2$	$-0-i\cdot\pi/2$	$-0-i\cdot\pi/2$	$+0-i\cdot\pi/2$	$+0-i\cdot\pi/2$	$+0-i\cdot\pi/2$	$+0-i\cdot\pi/2$
$+i\cdot\text{NAN}$	$-0+i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$	$-0+i\cdot\text{QNAN}$	$+0+i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$	$+0+i\cdot\text{QNAN}$	QNAN+i·QNAN

Notes:

- $\text{Atanh}(-1-i*0) = -\infty - i*0$, and ZERODIVIDE exception is raised
- raises INVALID exception when real or imaginary part of the argument is SNAN
- $\text{Atanh}(\text{CONJ}(z)) = \text{CONJ}(\text{Atanh}(z))$
- $\text{Atanh}(-z) = -\text{Atanh}(z)$.

Special Functions

v?Erf

Computes the error function value of vector elements.

Syntax

```
call vserf( n, a, y )
call vmserf( n, a, y, mode )
call vderf( n, a, y )
call vmderf( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vderf, vmderf REAL, INTENT(IN) for vserf, vmserf DOUBLE PRECISION, INTENT(IN) for vderf, vmderf	Array, specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vderf, vmderf REAL, INTENT(OUT) for vserf, vmserf DOUBLE PRECISION, INTENT(OUT) for vderf, vmderf	Array, specifies the output vector <i>y</i> .

Description

The `Erf` function computes the error function values for elements of the input vector a and writes them to the output vector y .

The error function is defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x) ,$

where `erfc` is the complementary error function.

2. $\Phi(x) = \frac{1}{2} \text{erf} (x/\sqrt{2}) ,$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp (- t^2/2) dt$$

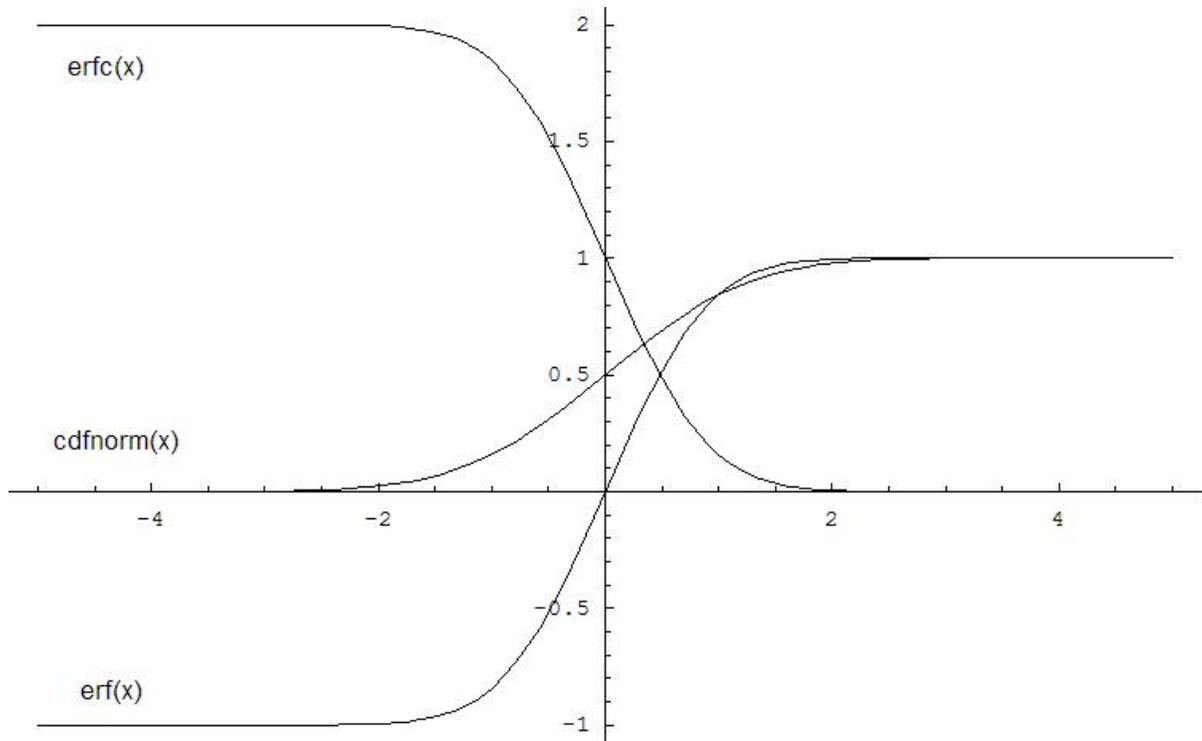
is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1) ,$

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

The following figure illustrates the relationships among `Erf` family functions (`Erf`, `Erfc`, `CdfNorm`).

Erf Family Functions Relationship



Useful relations for these functions:

$$\text{erf}(x) + \text{erfc}(x) = 1$$

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

Special Values for Real Function v?Erf(x)

Argument	Result	Exception
$+\infty$	+1	
$-\infty$	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See Also

- v?Erfc
- v?CdfNorm

v?Erfc

Computes the complementary error function value of vector elements.

Syntax

```
call vserfc( n, a, y )
call vmserfc( n, a, y, mode )
call vderfc( n, a, y )
call vmderfc( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vderfc</code> , <code>vmderfc</code> REAL, INTENT (IN) for <code>vserfc</code> , <code>vmserfc</code> DOUBLE PRECISION, INTENT (IN) for <code>vderfc</code> , <code>vmderfc</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vderfc</code> , <code>vmderfc</code> REAL, INTENT (OUT) for <code>vserfc</code> , <code>vmserfc</code> DOUBLE PRECISION, INTENT (OUT) for <code>vderfc</code> , <code>vmderfc</code>	Array that specifies the output vector <i>y</i> .

Description

The `erfc` function computes the complementary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The complementary error function is defined as follows:

$$\text{erfc}(x) = 1 - \text{erf}(x)$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x)$.

2. $\Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2})$,

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

$$3. \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

See also [Figure "Erf Family Functions Relationship"](#) in Erf function description for Erfc function relationship with the other functions of Erf family.

Special Values for Real Function v?Erfc(x)

Argument	Result	VM Error Status	Exception
X > underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	+0		
$-\infty$	+2		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

v?Erf
v?CdfNorm

v?CdfNorm

Computes the cumulative normal distribution function values of vector elements.

Syntax

```
call vscdfnorm( n, a, y )
call vmscdfnorm( n, a, y, mode )
call vdcdfnorm( n, a, y )
call vmdcdfnorm( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdcdfnorm, vmdcdfnorm REAL, INTENT(IN) for vscdfnorm, vmscdfnorm DOUBLE PRECISION, INTENT(IN) for vdcdfnorm, vmdcdfnorm	Array that specifies the input vector <i>a</i> .

Name	Type	Description
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vdcdfnorm</i> , <i>vmcdfnorm</i> REAL, INTENT (OUT) for <i>vscdfnorm</i> , <i>vmcdfnorm</i> DOUBLE PRECISION, INTENT (OUT) for <i>vdcdfnorm</i> , <i>vmcdfnorm</i>	Array that specifies the output vector <i>y</i> .

Description

The `CdfNorm` function computes the cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The cumulative normal distribution function is defined as given by:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

Useful relations:

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

where `Erf` and `Erfc` are the error and complementary error functions.

See also [Figure "Erf Family Functions Relationship"](#) in `Erf` function description for `CdfNorm` function relationship with the other functions of `Erf` family.

Special Values for Real Function v?CdfNorm(x)

Argument	Result	VM Error Status	Exception
$X < \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	+1		
$-\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?Erf](#)

[v?Erfc](#)

v?ErfInv

Computes inverse error function value of vector elements.

Syntax

```
call vserfinv( n, a, y )
call vmserfinv( n, a, y, mode )
call vderfinv( n, a, y )
call vmderfinv( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vderfinv, vmderfinv REAL, INTENT(IN) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(IN) for vderfinv, vmderfinv	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vderfinv, vmderfinv REAL, INTENT(OUT) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(OUT) for vderfinv, vmderfinv	Array that specifies the output vector <i>y</i> .

Description

The `ErfInv` function computes the inverse error function values for elements of the input vector *a* and writes them to the output vector *y*

$$y = \text{erf}^{-1}(a),$$

where `erf(x)` is the error function defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

1. $\text{erf}^{-1}(x) = \text{erfc}^{-1}(1 - x),$

where `erfc` is the complementary error function.

2. $\Phi(x) = \frac{1}{2} \text{erf} (x/\sqrt{2}),$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

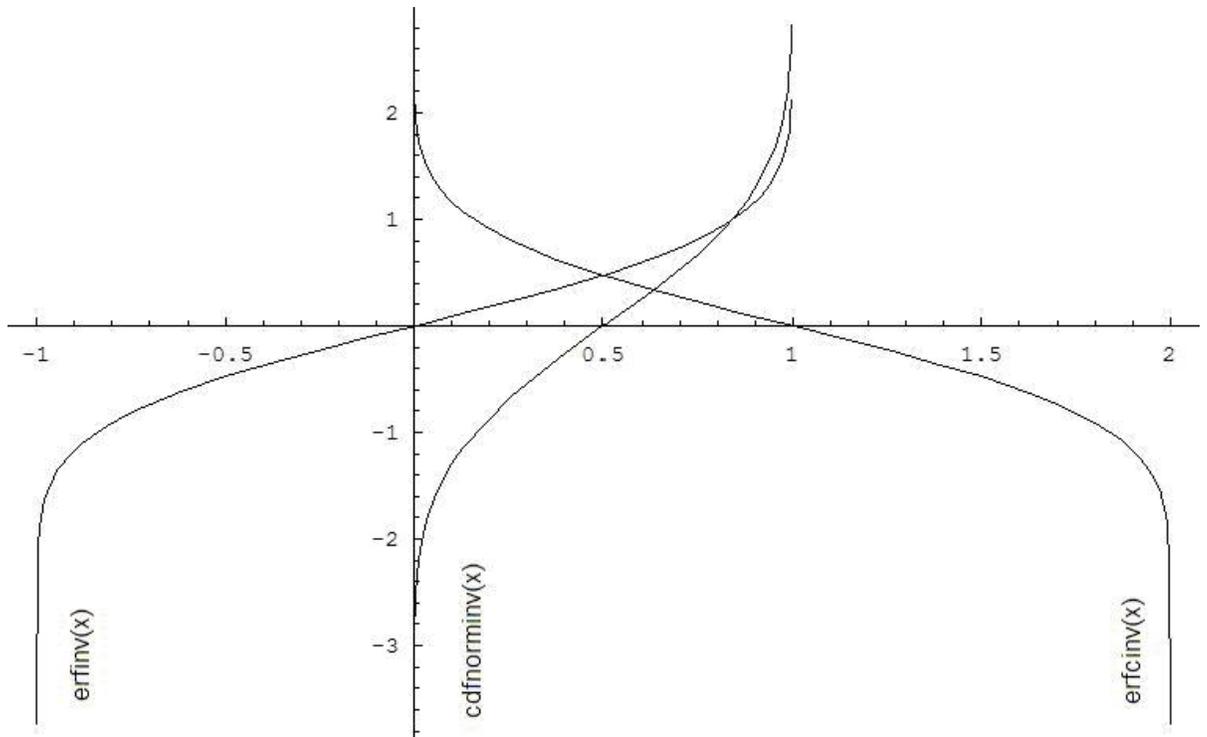
is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1),$

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

Figure "ErfInv Family Functions Relationship" illustrates the relationships among `ErfInv` family functions (`ErfInv`, `ErfcInv`, `CdfNormInv`).

ErfInv Family Functions Relationship



Useful relations for these functions:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

Special Values for Real Function v?ErfInv(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?ErfcInv](#)

[v?CdfNormInv](#)

v?ErfcInv

Computes the inverse complementary error function value of vector elements.

Syntax

```
call vserfcinv( n, a, y )
```

```
call vmserfcinv( n, a, y, mode )
```

```
call vderfcinv( n, a, y )
```

```
call vmderfcinv( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vderfcinv, vmderfcinv REAL, INTENT(IN) for vserfcinv, vmserfcinv	Array that specifies the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vderfcinv, vmderfcinv	
mode	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vderfcinv, vmderfcinv REAL, INTENT (OUT) for vserfcinv, vmserfcinv DOUBLE PRECISION, INTENT (OUT) for vderfcinv, vmderfcinv	Array that specifies the output vector y.

Description

The `ErfcInv` function computes the inverse complimentary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse complimentary error function is defined as given by:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where `erf(x)` denotes the error function and `erfinv(x)` denotes the inverse error function.

See also [Figure "ErfInv Family Functions Relationship"](#) in `ErfInv` function description for `ErfcInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function v?ErfcInv(x)

Argument	Result	VM Error Status	Exception
+1	+0		
+2	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
X < -0	QNAN	VML_STATUS_ERRDOM	INVALID
X > +2	QNAN	VML_STATUS_ERRDOM	INVALID

Argument	Result	VM Error Status	Exception
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?ErfInv](#)

[v?CdfNormInv](#)

v?CdfNormInv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

```
call vscdfnorminv( n, a, y )
call vmcdfnorminv( n, a, y, mode )
call vdcdfnorminv( n, a, y )
call vmdcdfnorminv( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdcdfnorminv, vmdcdfnorminv REAL, INTENT (IN) for vscdfnorminv, vmcdfnorminv DOUBLE PRECISION, INTENT (IN) for vdcdfnorminv, vmdcdfnorminv	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdcdfnorminv, vmdcdfnorminv REAL, INTENT (OUT) for vscdfnorminv, vmcdfnorminv DOUBLE PRECISION, INTENT (OUT) for vdcdfnorminv, vmdcdfnorminv	Array that specifies the output vector <i>y</i> .

Description

The `CdfNormInv` function computes the inverse cumulative normal distribution function values for elements of the input vector a and writes them to the output vector y .

The inverse cumulative normal distribution function is defined as given by:

$$\text{CdfNormInv}(x) = \text{CdfNorm}^{-1}(x) ,$$

where $\text{CdfNorm}(x)$ denotes the cumulative normal distribution function.

Useful relations:

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

where $\text{erfinv}(x)$ denotes the inverse error function and $\text{erfcinv}(x)$ denotes the inverse complementary error functions.

See also [Figure "ErfInv Family Functions Relationship"](#) in `ErfInv` function description for `CdfNormInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function v?CdfNormInv(x)

Argument	Result	VM Error Status	Exception
+0.5	+0		
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -0$	QNAN	VML_STATUS_ERRDOM	INVALID
$X > +1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?ErfInv](#)

[v?ErfcInv](#)

v?LGamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

```
call vslgamma( n, a, y )
```

```
call vmslgamma( n, a, y, mode )
```

```
call vdlgamma( n, a, y )
```

```
call vmdlgamma( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for <code>vdlgamma</code> , <code>vmdlgamma</code> REAL, INTENT (IN) for <code>vslgamma</code> , <code>vmslgamma</code> DOUBLE PRECISION, INTENT (IN) for <code>vdlgamma</code> , <code>vmdlgamma</code>	Array that specifies the input vector a .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdlgamma</code> , <code>vmdlgamma</code> REAL, INTENT (OUT) for <code>vslgamma</code> , <code>vmslgamma</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdlgamma</code> , <code>vmdlgamma</code>	Array that specifies the output vector y .

Description

The `v?LGamma` function computes the natural logarithm of the absolute value of gamma function for elements of the input vector a and writes them to the output vector y . Precision overflow thresholds for the `v?LGamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Special Values for Real Function `v?LGamma(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
+2	+0		
+0	+?	VML_STATUS_SING	ZERODIVIDE
-0	+?	VML_STATUS_SING	ZERODIVIDE
negative integer	+?	VML_STATUS_SING	ZERODIVIDE
-?	+?		
+?	+?		
X > overflow	+?	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		
SNAN	QNAN		INVALID

v?TGamma

Computes the gamma function of vector elements.

Syntax

```
call vstgamma( n, a, y )
call vmstgamma( n, a, y, mode )
call vdtgamma( n, a, y )
call vmdtgamma( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdtgamma, vmdtgamma REAL, INTENT (IN) for vstgamma, vmstgamma DOUBLE PRECISION, INTENT (IN) for vdtgamma, vmdtgamma	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdtgamma, vmdtgamma REAL, INTENT (OUT) for vstgamma, vmstgamma DOUBLE PRECISION, INTENT (OUT) for vdtgamma, vmdtgamma	Array that specifies the output vector <i>y</i> .

Description

The v?TGamma function computes the gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the v?TGamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW.

Special Values for Real Function v?TGamma(x)

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE

Argument	Result	VM Error Status	Exception
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
negative integer	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		
SNAN	QNAN		INVALID

Rounding Functions

v?Floor

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

```
call vsfloor( n, a, y )
```

```
call vmsfloor( n, a, y, mode )
```

```
call vdfloor( n, a, y )
```

```
call vmdfloor( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdfloor, vmdfloor REAL, INTENT(IN) for vsfloor, vmsfloor DOUBLE PRECISION, INTENT(IN) for vdfloor, vmdfloor	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdfloor, vmdfloor	Array that specifies the output vector <i>y</i> .

Name	Type	Description
	REAL, INTENT(OUT) for vsfloor, vmsfloor	
	DOUBLE PRECISION, INTENT(OUT) for vdfloor, vmdfloor	

Description

The function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Special Values for Real Function v?Floor(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Ceil

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

```
call vsceil( n, a, y )
call vmsceil( n, a, y, mode )
call vdceil( n, a, y )
call vmdceil( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdceil, vmdceil REAL, INTENT(IN) for vsceil, vmsceil DOUBLE PRECISION, INTENT(IN) for vdfloor, vmdfloor	Array that specifies the input vector <i>a</i> .

Name	Type	Description
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <i>vdceil</i> , <i>vmdceil</i> REAL, INTENT (OUT) for <i>vsceil</i> , <i>vmsceil</i> DOUBLE PRECISION, INTENT (OUT) for <i>vdceil</i> , <i>vmdceil</i>	Array that specifies the output vector <i>y</i> .

Description

The function computes an integer value rounded towards plus infinity for each vector element.

$$y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Ceil(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

```
call vstrunc( n, a, y )
call vmstrunc( n, a, y, mode )
call vdtrunc( n, a, y )
call vmdtrunc( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for vdTrunc, vmdTrunc REAL, INTENT (IN) for vsTrunc, vmsTrunc DOUBLE PRECISION, INTENT (IN) for vdTrunc, vmdTrunc	Array that specifies the input vector a .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for vdTrunc, vmdTrunc REAL, INTENT (OUT) for vsTrunc, vmsTrunc DOUBLE PRECISION, INTENT (OUT) for vdTrunc, vmdTrunc	Array that specifies the output vector y .

Description

The function computes an integer value rounded towards zero for each vector element.

$$a_i \geq 0, y_i = \lfloor a_i \rfloor$$

$$a_i < 0, y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Trunc(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Round

Computes a value rounded to the nearest integer for each vector element.

Syntax

```
call vsround( n, a, y )
```

```
call vmsround( n, a, y, mode )
```

```
call vdround( n, a, y )
```

```
call vmdround( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for <code>vdround</code> , <code>vmdround</code> REAL, INTENT (IN) for <code>vsround</code> , <code>vmsround</code> DOUBLE PRECISION, INTENT (IN) for <code>vdround</code> , <code>vmdround</code>	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for <code>vdround</code> , <code>vmdround</code> REAL, INTENT (OUT) for <code>vsround</code> , <code>vmsround</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdround</code> , <code>vmdround</code>	Array that specifies the output vector <i>y</i> .

Description

The function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Special Values for Real Function `v?Round(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID

Argument	Result	Exception
QNaN	QNaN	

v?NearbyInt

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

```
call vsnearbyint( n, a, y )
call vmsnearbyint( n, a, y, mode )
call vdnearbyint( n, a, y )
call vmdnearbyint( n, a, y, mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdnearbyint, vmdnearbyint REAL, INTENT(IN) for vsnearbyint, vmsnearbyint DOUBLE PRECISION, INTENT(IN) for vdnearbyint, vmdnearbyint	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER(KIND=8), INTENT(IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdnearbyint, vmdnearbyint REAL, INTENT(OUT) for vsnearbyint, vmsnearbyint DOUBLE PRECISION, INTENT(OUT) for vdnearbyint, vmdnearbyint	Array that specifies the output vector <i>y</i> .

Description

The `v?NearbyInt` function computes a rounded integer value in a current rounding mode for each vector element.

Special Values for Real Function $v?NearbyInt(x)$

Argument	Argument	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

 $v?Rint$

Computes a rounded integer value in the current rounding mode.

Syntax

```
call vsrint( n, a, y )
call vmsrint( n, a, y, mode )
call vdrint( n, a, y )
call vmdrint( n, a, y, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdrint, vmdrint REAL, INTENT (IN) for vsrint, vmsrint DOUBLE PRECISION, INTENT (IN) for vdrint, vmdrint	Array that specifies the input vector <i>a</i> .
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdrint, vmdrint REAL, INTENT (OUT) for vsrint, vmsrint DOUBLE PRECISION, INTENT (OUT) for vdrint, vmdrint	Array that specifies the output vector <i>y</i> .

Description

The `v?Rint` function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- $f(0.5) = 0$, for rounding modes set to round to nearest round toward zero or to minus infinity.
- $f(0.5) = 1$, for rounding modes set to plus infinity.
- $f(-1.5) = -2$, for rounding modes set to round to nearest or to minus infinity.
- $f(-1.5) = -1$, for rounding modes set to round toward zero or to plus infinity.

Special Values for Real Function `v?Rint(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

`v?Modf`

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

```
call vsmodf( n, a, y, z )
```

```
call vmsmodf( n, a, y, z, mode )
```

```
call vdmodf( n, a, y, z )
```

```
call vmdmodf( n, a, y, z, mode )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdmodf</code> , <code>vmdmodf</code> REAL, INTENT (IN) for <code>vsmodf</code> , <code>vmsmodf</code> DOUBLE PRECISION, INTENT (IN) for <code>vdmodf</code> , <code>vmdmodf</code>	Array, specifies the input vector <code>a</code> .
<code>mode</code>	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, z	DOUBLE PRECISION for <code>vdmodf</code> , <code>vmdmodf</code> REAL, INTENT(OUT) for <code>vsmodf</code> , <code>vmsmodf</code> DOUBLE PRECISION, INTENT(OUT) for <code>vdmodf</code> , <code>vmdmodf</code>	Array, specifies the output vector y and z .

Description

The function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Special Values for Real Function `v?Modf(x)`

Argument	Result: $y(i)$	Result: $z(i)$	Exception
+0	+0	+0	
-0	-0	-0	
$+\infty$	$+\infty$	+0	
$-\infty$	$-\infty$	-0	
SNAN	QNAN	QNAN	INVALID
QNAN	QNAN	QNAN	

`v?Frac`

Computes a signed fractional part for each vector element.

Syntax

```
call vsfrac( n, a, y )
call vmsfrac( n, a, y, mode )
call vdfrac( n, a, y )
call vmdfrac( n, a, y, mode )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
n	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
a	DOUBLE PRECISION for <code>vdfrac</code> , <code>vmdfrac</code> REAL, INTENT (IN) for <code>vsfrac</code> , <code>vmsfrac</code> DOUBLE PRECISION, INTENT (IN) for <code>vdfrac</code> , <code>vmdfrac</code>	Array that specifies the input vector a .
$mode$	INTEGER (KIND=8), INTENT (IN)	Overrides global VM mode setting for this function call. See <code>vm1SetMode</code> for possible values and their description.

Output Parameters

Name	Type	Description
y	DOUBLE PRECISION for <code>vdfrac</code> , <code>vmdfrac</code> REAL, INTENT (OUT) for <code>vsfrac</code> , <code>vmsfrac</code> DOUBLE PRECISION, INTENT (OUT) for <code>vdfrac</code> , <code>vmdfrac</code>	Array that specifies the output vector y .

Description

The function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, & a_i \geq 0 \\ a_i - \lceil a_i \rceil, & a_i < 0 \end{cases}$$

Special Values for Real Function `v?Frac(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	+0	
$-\infty$	-0	
SNAN	QNAN	INVALID
QNAN	QNAN	

VM Pack/Unpack Functions

This section describes VM functions that convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix B for details on vector indexing methods).

The table below lists available VM Pack/Unpack functions, together with data types and indexing methods associated with them.

VM Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
<code>v?pack</code>	s, d, c, z	I,V,M	Gathers elements of arrays, indexed by different methods.
<code>v?unpack</code>	s, d, c, z	I,V,M	Scatters vector elements to arrays with different indexing.

See Also

Vector Arguments in VM

v?Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

```
call vspacki( n, a, inca, y )
call vspackv( n, a, ia, y )
call vspackm( n, a, ma, y )
call vdpacki( n, a, inca, y )
call vdpackv( n, a, ia, y )
call vdpackm( n, a, ma, y )
call vcpacki( n, a, inca, y )
call vcpackv( n, a, ia, y )
call vcpackm( n, a, ma, y )
call vzpacki( n, a, inca, y )
call vzpackv( n, a, ia, y )
call vzpackm( n, a, ma, y )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Type	Description
<code>n</code>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<code>a</code>	DOUBLE PRECISION for <code>vdpacki</code> , <code>vdpackv</code> , <code>vdpackm</code> COMPLEX for <code>vcpacki</code> , <code>vcpackv</code> , <code>vcpackm</code> DOUBLE COMPLEX for <code>vzpacki</code> , <code>vzpackv</code> , <code>vzpackm</code>	Array, DIMENSION at least $(1 + (n-1)*inca)$ for <code>v?packi</code> , Array, DIMENSION at least $\max(n, \max(ia[j]))$, $j=0, \dots, n-1$ for <code>v?packv</code> , Array, DIMENSION at least n for <code>v?packm</code> . Specifies the input vector <code>a</code> .

Name	Type	Description
	REAL, INTENT (IN) for vspacki, vspackv, vspackm DOUBLE PRECISION, INTENT (IN) for vdpacki, vdpackv, vdpackm COMPLEX, INTENT (IN) for vcpacki, vcpackv, vcpackm DOUBLE COMPLEX, INTENT (IN) for vzpacki, vzpackv, vzpackm	
<i>inca</i>	INTEGER, INTENT (IN) for vspacki, vdpacki, vcpacki, vzpacki	Specifies the increment for the elements of <i>a</i> .
<i>ia</i>	FORTRAN 77: INTEGER for vspackv, vdpackv, vcpackv, vzpackv INTEGER, INTENT (IN) for vspackv, vdpackv, vcpackv, vzpackv	Array, DIMENSION at least <i>n</i> . Specifies the index vector for the elements of <i>a</i> .
<i>ma</i>	FORTRAN 77: INTEGER for vspackm, vdpackm, vcpackm, vzpackm Fortran 90: INTEGER, INTENT (IN) for vspackm, vdpackm, vcpackm, vzpackm	Array, DIMENSION at least <i>n</i> , Specifies the mask vector for the elements of <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdpacki, vdpackv, vdpackm COMPLEX for vcpacki, vcpackv, vcpackm DOUBLE COMPLEX for vzpacki, vzpackv, vzpackm REAL, INTENT (OUT) for vspacki, vspackv, vspackm DOUBLE PRECISION, INTENT (OUT) for vdpacki, vdpackv, vdpackm COMPLEX, INTENT (OUT) for vcpacki, vcpackv, vcpackm DOUBLE COMPLEX, INTENT (OUT) for vzpacki, vzpackv, vzpackm	Array, DIMENSION at least <i>n</i> . Specifies the output vector <i>y</i> .

v?Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
call vcunpacki( n, a, y, incy )
call vcunpackv( n, a, y, iy )
call vcunpackm( n, a, y, my )
call vzunpacki( n, a, y, incy )
call vzunpackv( n, a, y, iy )
call vzunpackm( n, a, y, my )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Specifies the number of elements to be calculated.
<i>a</i>	DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm COMPLEX for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX for vzunpacki, vzunpackv, vzunpackm REAL, INTENT (IN) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT (IN) for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT (IN) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT (IN) for vzunpacki, vzunpackv, vzunpackm	Array, DIMENSION at least <i>n</i> . Specifies the input vector <i>a</i> .
<i>incy</i>	INTEGER, INTENT (IN) for vsunpacki, vdunpacki, vcunpacki, vzunpacki	Specifies the increment for the elements of <i>y</i> .

Name	Type	Description
<i>iy</i>	INTEGER, INTENT (IN) for vsunpackv, vdunpackv, vcunpackv, vzunpackv	Array, DIMENSION at least <i>n</i> . Specifies the index vector for the elements of <i>y</i> .
<i>my</i>	INTEGER, INTENT (IN) for vsunpackm, vdunpackm, vcunpackm, vzunpackm	Array, DIMENSION at least <i>n</i> , Specifies the mask vector for the elements of <i>y</i> .

Output Parameters

Name	Type	Description
<i>y</i>	DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT (IN) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT (IN) for vzunpacki, vzunpackv, vzunpackm REAL, INTENT (OUT) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT (OUT) for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT (OUT) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT (OUT) for vzunpacki, vzunpackv, vzunpackm	Array, DIMENSION for v?unpacki, at least $(1 + (n-1) * incy)$ for v?unpackv, at least $\max(n, \max(iy[j]), j=0, \dots, n-1)$ for v?unpackm, at least <i>n</i> for v?UnpackI, at least $(1 + (n-1) * incy)$ for v?UnpackV, at least $\max(n, \max(ia[j]), j=0, \dots, n-1)$ for v?UnpackM, at least <i>n</i> .

VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

VM Service Functions

Function Short Name	Description
vmlsetmode	Sets the VM mode
vmlgetmode	Gets the VM mode
vmlseterrstatus	Sets the VM Error Status
vmlgeterrstatus	Gets the VM Error Status
vmlclearerrstatus	Clears the VM Error Status
vmlseterrorcallback	Sets the additional error handler callback function
vmlgeterrorcallback	Gets the additional error handler callback function
vmlclearerrorcallback	Deletes the additional error handler callback function

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice

effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

vmlSetMode

Sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*.

Syntax

```
oldmode = vmlsetmode( mode )
```

Include Files

- mkl_vml.f90

Input Parameters

Name	Type	Description
<i>mode</i>	INTEGER (KIND=8), INTENT (IN)	Specifies the VM mode to be set.

Output Parameters

Name	Type	Description
<i>oldmode</i>	INTEGER*8 INTEGER (KIND=8)	Specifies the former VM mode.

Description

The `vmlSetMode` function sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*. The mode change has a global effect on all the VM functions within a thread.

NOTE

You can override the global mode setting and change the mode for a given VM function call by using a respective `vm[s, d]<Func>` variant of the function.

The *mode* parameter is designed to control accuracy, handling of denormalized numbers, and error handling. [Table "Values of the *mode* Parameter"](#) lists values of the *mode* parameter. You can obtain all other possible values of the *mode* parameter from the *mode* parameter values by using bitwise OR (|) operation to combine one value for accuracy, one value for handling of denormalized numbers, and one value for error control options. The default value of the *mode* parameter is `VML_HA | VML_FTZDAZ_OFF | VML_ERRMODE_DEFAULT`.

The `VML_FTZDAZ_ON` mode is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (DAZ = denormals-are-zero) and denormalized results are flushed to zero (FTZ = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

Values of the *mode* Parameter

Value of <i>mode</i>	Description
Accuracy Control	
VML_HA	high accuracy versions of VM functions
VML_LA	low accuracy versions of VM functions
VML_EP	enhanced performance accuracy versions of VM functions
Denormalized Numbers Handling Control	
VML_FTZDAZ_ON	Faster processing of denormalized inputs is enabled.
VML_FTZDAZ_OFF	Faster processing of denormalized inputs is disabled.
Error Mode Control	
VML_ERRMODE_IGNORE	No action is set for computation errors.
VML_ERRMODE_STDERR	On error, the error text information is written to <i>stderr</i> .
VML_ERRMODE_EXCEPT	On error, an exception is raised.
VML_ERRMODE_CALLBACK	On error, an additional error handler function is called.
VML_ERRMODE_DEFAULT	On error, an exception is raised and an additional error handler function is called.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

The following example shows how to set low accuracy, fast processing for denormalized numbers and *stderr* error mode:

```
oldmode = vmlsetmode( VML_LA )
call vmlsetmode( IOR(VML_LA, VML_FTZDAZ_ON, VML_ERRMODE_STDERR) )
```

vmlgetmode

Gets the VM mode.

Syntax

```
mod = vmlgetmode()
```

Include Files

- `mkl_vml.f90`

Output Parameters

Name	Type	Description
<i>mod</i>	INTEGER	Specifies the packed <i>mode</i> parameter.

Description

The function `vmlgetmode` returns the VM *mode* parameter that controls accuracy, handling of denormalized numbers, and error handling options. The *mod* variable value is a combination of the values listed in the table "Values of the *mode* Parameter". You can obtain these values using the respective mask from the table "Values of Mask for the *mode* Parameter".

Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy <i>mode</i> selection.
VML_FTZDAZ_MASK	Specifies mask for FTZDAZ <i>mode</i> selection.
VML_ERRMODE_MASK	Specifies mask for error <i>mode</i> selection.

See example below:

Examples

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
denm = IAND(mod, VML_FTZDAZ_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)
```

vmlSetErrStatus

Sets the new VM Error Status according to *err* and stores the previous VM Error Status to *olderr*.

Syntax

```
olderr = vmlseterrstatus( status )
```

Include Files

- `mk1_vml.f90`

Input Parameters

Name	Type	Description
<i>status</i>	INTEGER, INTENT (IN)	Specifies the VM error status to be set.

Output Parameters

Name	Type	Description
<i>olderr</i>	INTEGER	Specifies the former VM error status.

Description

Table "Values of the VM Status" lists possible values of the *err* parameter.

Values of the VM Status

Status	Description
Successful Execution	
VML_STATUS_OK	The execution was completed successfully.
Warnings	
VML_STATUS_ACCURACYWARNING	The execution was completed successfully in a different accuracy mode.
Errors	
VML_STATUS_BADSIZE	The function does not support the preset accuracy mode. The Low Accuracy mode is used instead.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of the input array values causes a divide-by-zero exception or produces an invalid (QNaN) result.
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
olderr = vmlSetErrStatus( VML_STATUS_OK );
```

```
olderr = vmlSetErrStatus( VML_STATUS_ERRDOM );
```

```
olderr = vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

vmlgeterrstatus

Gets the VM Error Status.

Syntax

```
err = vmlgeterrstatus( )
```

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
<i>err</i>	INTEGER	Specifies the VM error status.

vmlclearerrstatus

Sets the VM Error Status to VML_STATUS_OK and stores the previous VM Error Status to olderr.

Syntax

```
olderr = vmlclearerrstatus( )
```

Include Files

- `mkl_vml.f90`

Output Parameters

Name	Type	Description
<code>olderr</code>	INTEGER	Specifies the former VM error status.

vmlSetErrorCallback

Sets the additional error handler callback function and gets the old callback function.

Syntax

```
oldcallback = vmlseterrorcallback( callback )
```

Include Files

- `mkl_vml.f90`

Input Parameters

Name	Description
<code>callback</code>	Address of the callback function.

Description
The callback function has the following format:
<code>INTEGER FUNCTION ERRFUNC(par)</code>
<code>TYPE (ERROR_STRUCTURE) par</code>
<code>! ...</code>
<code>! user error processing</code>
<code>! ...</code>
<code>ERRFUNC = 0</code>
<code>! if ERRFUNC= 0 - standard VM error handler</code>
<code>! is called after the callback</code>
<code>! if ERRFUNC != 0 - standard VM error handler</code>
<code>! is not called</code>
<code>END</code>

Name

Description

The passed error structure is defined as follows:

```
TYPE ERROR_STRUCTURE SEQUENCE
```

```
INTEGER*4 ICODE
```

```
INTEGER*4 IINDEX
```

```
REAL*8 DBA1
```

```
REAL*8 DBA2
```

```
REAL*8 DBR1
```

```
REAL*8 DBR2
```

```
CHARACTER(64) CFUNCNAME
```

```
INTEGER*4 IFUNCNAMELEN
```

```
REAL*8 DBA1IM
```

```
REAL*8 DBA2IM
```

```
REAL*8 DBR1IM
```

```
REAL*8 DBR2IM
```

```
END TYPE ERROR_STRUCTURE
```

Output Parameters

Name

Type

Description

oldcallback

INTEGER

Address of the former callback function.

NOTE

This function does not have a FORTRAN 77 interface due to the use of internal structures.

Description

The callback function is called on each VM mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see "Values of the *mode* Parameter").

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

vmlGetErrorCallBack

Gets the additional error handler callback function.

Syntax

```
callback = vmlgeterrorcallback( )
```

Include Files

- mkl_vml.f90

Output Parameters

Name	Description
<i>callback</i>	Address of the callback function

vmlClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

```
oldcallback = vmlclearerrorcallback( )
```

Include Files

- mkl_vml.f90

Output Parameters

Name	Type	Description
<i>oldcallback</i>	INTEGER	Address of the former callback function

Statistical Functions

Statistical functions in Intel® MKL are known as the Vector Statistics (VS). They are designed for the purpose of

- generating vectors of pseudorandom, quasi-random, and non-deterministic random numbers
- performing mathematical operations of convolution and correlation
- computing basic statistical estimates for single and double precision multi-dimensional datasets

The corresponding functionality is described in the respective [Random Number Generators](#), [Convolution and Correlation](#), and [Summary Statistics](#) sections.

See VS performance data in the online VS Performance Data document available at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>

The basic notion in VS is a task. The task object is a data structure or descriptor that holds the parameters related to a specific statistical operation: random number generation, convolution and correlation, or summary statistics estimation. Such parameters can be an identifier of a random number generator, its internal state and parameters, data arrays, their shape and dimensions, an identifier of the operation and so forth. You can modify the VS task parameters using the VS service functions.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Random Number Generators

Intel MKL VS provides a set of routines implementing commonly used pseudorandom, quasi-random, or non-deterministic random number generators with continuous and discrete distribution. To improve performance, all these routines were developed using the calls to the highly optimized *Basic Random Number Generators* (BRNGs) and vector mathematical functions (VM, see [Chapter 9, "Vector Mathematical Functions"](#)).

VS provides interfaces both for Fortran and C languages. For users of the Fortran 90 or Fortran 95 language the `mkl_vs1.f90` header file is provided. The `mkl_vs1.fi` header file available in the previous versions of Intel MKL is retained for backward compatibility. All header files are found in the following directory:

```
$(MKL)/include
```

The `mkl_vs1.f90` header is intended for use with the Fortran `include` clause and is compatible with both standard forms of F90/F95 sources - the free and 72-columns fixed forms. If you need to use the VS interface with 80- or 132-columns fixed form sources, you may add a new file to your project. That file is formatted as a 72-columns fixed-form source and consists of a single `include` clause as follows:

```
include 'mkl_vs1.f90'
```

This `include` clause causes the compiler to generate the module files `mkl_vsl.mod` and `mkl_vsl_type.mod`, which are used to process the Fortran use clauses referencing to the VS interface:

```
use mkl_vsl_type
```

```
use mkl_vsl
```

Because of this specific feature, you do not need to include the `mkl_vsl.f90` header into each source of your project. You only need to include the header into some of the sources. In any case, make sure that the sources that depend on the VS interface are compiled after those that include the header so that the module files `mkl_vsl.mod` and `mkl_vsl_type.mod` are generated prior to using them.

NOTE

For the Fortran interface, VS provides both a subroutine-style interface and a function-style interface. The default interface in this case is a function-style interface. The function-style interface, unlike the subroutine-style interface, allows the user to get error status of each routine. The subroutine-style interface is provided for backward compatibility only. To use the subroutine-style interface, manually include `mkl_vsl_subroutine.fi` file instead of `mkl_vsl.f90` by changing the line `include 'mkl_vsl.f90'` in `include\mkl.fi` with the line `include 'mkl_vsl_subroutine.fi'`.

All VS routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are pseudorandom, quasi-random, or non-deterministic random number generators. Detailed description of the generators can be found in [Distribution Generators](#) section.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in [Service Routines](#) section.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see [Advanced Service Routines](#) section).

The last two categories are referred to as service routines.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to the '*Random Numbers*' section in [VS Notes](#) document provided at the Intel® MKL web page.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See [VS Notes](#) for the description of methods available for each generator.

An RNG task determines environment in which random number generation is performed, in particular parameters of the BRNG and its internal state. Output of VS generators is a stream of random numbers that are used in Monte Carlo simulations. A *random stream descriptor* and a *random stream* are used as synonyms of an *RNG task* in the document unless the context requires otherwise.

The *random stream descriptor* specifies which BRNG should be used in a given transformation method. See the *Random Streams and RNGs in Parallel Computation* section of [VS Notes](#).

The term *computational node* means a logical or physical unit that can process data in parallel.

Mathematical Notation

The following notation is used throughout the text:

- N The set of natural numbers $N = \{1, 2, 3 \dots\}$.
- Z The set of integers $Z = \{\dots -3, -2, -1, 0, 1, 2, 3 \dots\}$.
- R The set of real numbers.

$\lfloor a \rfloor$ The floor of a (the largest integer less than or equal to a).

\oplus or **xor** Bitwise exclusive OR.

C_{α}^{κ} or $\binom{\alpha}{\kappa}$ Binomial coefficient or combination ($\alpha \in R, \alpha \geq 0; \kappa \in N \cup \{0\}$).

$$C_{\alpha}^0 = 1$$

For $\alpha \geq k$ binomial coefficient is defined as

$$C_{\alpha}^{\kappa} = \frac{\alpha(\alpha - 1) \dots (\alpha - \kappa + 1)}{\kappa!}$$

If $\alpha < k$, then

$$C_{\alpha}^{\kappa} = 0$$

$\Phi(x)$ Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$

defined over $-\infty < x < +\infty$.

$\Phi(-\infty) = 0, \Phi(+\infty) = 1$.

$\Gamma(\alpha)$ The complete gamma function

$$\Gamma(\alpha) = \int_0^{\infty} t^{\alpha-1} e^{-t} dt$$

where $\alpha > 0$.

$B(p, q)$

The complete beta function

$$B(p, q) = \int_0^1 t^{p-1} (1 - t)^{q-1} dt$$

where $p > 0$ and $q > 0$.

LCG(a, c, m)

Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the *multiplier*, c is called the *increment*, and m is called the *modulus* of the generator.

MCG(a, m)

Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.

GFSR(p, q)

Generalized Feedback Shift Register Generator

$$x_n = x_{n-p} \oplus x_{n-q}$$

Naming Conventions

The names of the routines in VS random number generators are lowercase (*virnguniform*). The names are not case-sensitive.

The names of generator routines have the following structure:

v<type of result>rng<distribution>

where

- *v* is the prefix of a VS vector function.
- <type of result> is either *s*, *d*, or *i* and specifies one of the following types:

<i>s</i>	REAL
<i>d</i>	DOUBLE PRECISION
<i>i</i>	INTEGER

Prefixes *s* and *d* apply to continuous distributions only, prefix *i* applies only to discrete case.

- *rng* indicates that the routine is a random generator.
- <distribution> specifies the type of statistical distribution.

Names of service routines follow the template below:

vsl<name>

where

- *vsl* is the prefix of a VS service function.

- `<name>` contains a short function name.

For a more detailed description of service routines, refer to [Service Routines](#) and [Advanced Service Routines](#) sections.

The prototype of each generator routine corresponding to a given probability distribution fits the following structure:

```
status = <function name>( method, stream, n, r, [<distribution parameters>] )
```

where

- `method` defines the method of generation. A detailed description of this parameter can be found in table "[Values of <method> in method parameter](#)". See the next page, where the structure of the `method` parameter name is explained.
- `stream` defines the descriptor of the random stream and must have a non-zero value. Random streams, descriptors, and their usage are discussed further in [Random Streams](#) and [Service Routines](#).
- `n` defines the number of random values to be generated. If `n` is less than or equal to zero, no values are generated. Furthermore, if `n` is negative, an error condition is set.
- `r` defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least `n` random numbers.
- `status` defines the error status of a VS routine. See [Error Reporting](#) section for a detailed description of error status values.

Additional parameters included into `<distribution parameters>` field are individual for each generator routine and are described in detail in [Distribution Generators](#) section.

To invoke a distribution generator, use a call to the respective VS routine. For example, to obtain a vector `r`, composed of `n` independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value `a` and standard deviation `sigma`, write the following:

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

The name of a `method` parameter has the following structure:

```
VSL_RNG_METHOD_method<distribution>_<method>
```

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

where

- `<distribution>` is the probability distribution.
- `<method>` is the method name.

Type of the name structure for the `method` parameter corresponds to fast and accurate modes of random number generation (see "[Distribution Generators](#)" section and [VS Notes](#) for details).

Method names `VSL_RNG_METHOD_<distribution>_<method>`

and

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

should be used with

```
v<precision>Rng<distribution>
```

function only, where

- `<precision>` is

<code>s</code>	for single precision continuous distribution
<code>d</code>	for double precision continuous distribution
<code>i</code>	for discrete distribution
- `<distribution>` is the probability distribution.

is the probability distribution. Table "[Values of <method> in method parameter](#)" provides specific predefined values of the `method` name. The third column contains names of the functions that use the given method.

Values of <method> in *method* parameter

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous), Uniform (discrete), UniformBits, UniformBits32, UniformBits64
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	Gaussian, GaussianMV
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$ $x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$	Gaussian, GaussianMV, Lognormal
ICDF	Inverse cumulative distribution function method.	Exponential, Laplace, Weibull, Cauchy, Rayleigh, Gumbel, Bernoulli, Geometric, Gaussian, GaussianMV, Lognormal
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	Gamma
CJA	For $\min(p, q) > 1$, Cheng method is used; for $\min(p, q) < 1$, Johnk method is used, if $q + K \cdot p^2 + C \leq 0$ ($K = 0.852\dots$, $C = -0.956\dots$) otherwise, Atkinson switching algorithm is used; for $\max(p, q) < 1$, method of Johnk is used; for $\min(p, q) < 1$, $\max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta

Method	Short Description	Functions
BTPE	Acceptance/rejection method for $n_{trial} \cdot \min(p, 1 - p) \geq 30$ with decomposition into 4 regions: <ul style="list-style-type: none"> - 2 parallelograms - triangle - left exponential tail - right exponential tail 	Binomial
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: <ul style="list-style-type: none"> - rectangular - left exponential tail - right exponential tail 	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: <ul style="list-style-type: none"> - 2 parallelograms - triangle - left exponential tail - right exponential tail; otherwise, table lookup method is used.	Poisson
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	Poisson , PoissonV
NBAR	Acceptance/rejection method for , $\frac{(a - 1) \cdot (1 - p)}{p} \geq 100$ with decomposition into 5 regions: <ul style="list-style-type: none"> - rectangular - 2 trapezoid - left exponential tail - right exponential tail 	NegBinomial

NOTE

In this document, routines are often referred to by their base name (`Gaussian`) when this does not lead to ambiguity. In the routine reference, the full name (`vsrnggaussian`, `vsRngGaussian`) is always used in prototypes and code examples.

Basic Generators

VS provides pseudorandom, quasi-random, and non-deterministic random number generators. This includes the following BRNGs, which differ in speed and other properties:

- the 31-bit multiplicative congruential pseudorandom number generator $MCG(1132489760, 2^{31} - 1)$ [L'Ecuyer99]
- the 32-bit generalized feedback shift register pseudorandom number generator $GFSR(250, 103)$ [Kirkpatrick81]
- the combined multiple recursive pseudorandom number generator $MRG32k3a$ [L'Ecuyer99a]
- the 59-bit multiplicative congruential pseudorandom number generator $MCG(13^{13}, 2^{59})$ from NAG Numerical Libraries [NAG]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]
- Mersenne Twister pseudorandom number generator $MT19937$ [Matsumoto98] with period length $2^{19937} - 1$ of the produced sequence
- Set of 6024 Mersenne Twister pseudorandom number generators $MT2203$ [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to $2^{2203} - 1$. Parameters of the generators provide mutual independence of the corresponding sequences.
- SIMD-oriented Fast Mersenne Twister pseudorandom number generator $SFMT19937$ [Saito08] with a period length equal to $2^{19937} - 1$ of the produced sequence.
- Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VS Notes](#).
- Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VS Notes](#).
- Non-deterministic random number generator (RDRAND-based generators only) [AVX], [IntelSWMan].

NOTE

You can use a non-deterministic random number generator only if the underlying hardware supports it. For instructions on how to detect if an Intel CPU supports a non-deterministic random number generator see, for example, *Chapter 8: Post-32nm Processor Instructions* in [AVX] or *Chapter 4: RdRand Instruction Usage* in [BMT].

NOTE

The time required by some non-deterministic sources to generate a random number is not constant, so you might have to make multiple requests before the next random number is available. VS limits the number of retries for requests to the non-deterministic source to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the `vslNewStreamEx` function.

For more details on the non-deterministic source implementation for Intel CPUs please refer to Section 7.3.17, Volume 1, *Random Number Generator Instruction* in [IntelSWMan] and Section 4.2.2, *RdRand Retry Loop* in [BMT].

- Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} $PHILOX4X32X10$ [Salmon11].
- ARS-5 counter-based pseudorandom number generator with a period of 2^{128} , which uses instructions from the AES-NI set $ARS5$ [Salmon11].

See some testing results for the generators in [VS Notes](#) and comparative performance data at <https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation>.

VS provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#) section.

For some basic generators, VS provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VS provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#) section.

There is also an option to utilize externally generated random numbers in VS distribution generator routines. For this purpose VS provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval.

Such basic generators are called the abstract basic random number generators.

See [VS Notes](#) for a more detailed description of the generator properties.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BRNG Parameter Definition

Predefined values for the *brng* input parameter are as follows:

Values of *brng* parameter

Value	Short Description
VSL_BRNG_MCG31	A 31-bit multiplicative congruential generator.
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MRG32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 6024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SFMT19937	A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.
VSL_BRNG_SOBOL	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 40$; user-defined dimensions are also available.

Value	Short Description
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.
VSL_BRNG_NONDETERM	A non-deterministic random number generator.
VSL_BRNG_PHILOX4X32X10	A Philox4x32-10 counter-based pseudorandom number generator.
VSL_BRNG_ARS5	An ARS-5 counter-based pseudorandom number generator that uses instructions from the AES-NI set.

See [VS Notes](#) for detailed description.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. You can operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See [VS Notes](#) for details.

NOTE

Random streams associated with abstract basic random number generator are called the abstract random streams. See [VS Notes](#) for detailed description of abstract streams and their use.

You can create unlimited number of random streams by VS [Service Routines](#) like [NewStream](#) and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine [DeleteStream](#).

VS provides service functions [SaveStreamF](#) and [LoadStreamF](#) to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See [VS Notes](#) for detailed description.

Data Types

FORTRAN 77:

```
INTEGER*4 vslstreamstate(2)
```

Fortran 90:

```
TYPEVSL_STREAM_STATEINTEGER*4 descriptor1INTEGER*4 descriptor2ENDTYPE VSL_STREAM_STATE
```

Error Reporting

VS RNG routines return status codes of the performed operation to report errors to the calling program. The application should perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

VSL_ERROR_<ERROR_NAME> - indicates VS errors common for all VS domains.

VSL_RNG_ERROR_<ERROR_NAME> - indicates VS RNG errors.

VS RNG errors are of negative values while warnings are of positive values. The status code of zero value indicates successful completion of the operation: VSL_ERROR_OK (or synonymic VSL_STATUS_OK).

Status Codes

Status Code	Description
Common VSL	
VSL_ERROR_OK, VSL_STATUS_OK	No error, execution is successful.
VSL_ERROR_BADARGS	Input argument value is not valid.
VSL_ERROR_CPU_NOT_SUPPORTED	CPU version is not supported.
VSL_ERROR_FEATURE_NOT_IMPLEMENTED	Feature invoked is not implemented.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory.
VSL_ERROR_NULL_PTR	Input pointer argument is NULL.
VSL_ERROR_UNKNOWN	Unknown error.
VS RNG Specific	
VSL_RNG_ERROR_BAD_FILE_FORMAT	File format is unknown.
VSL_RNG_ERROR_BAD_MEM_FORMAT	Descriptive random stream format is unknown.
VSL_RNG_ERROR_BAD_NBITS	The value in NBits field is bad.
VSL_RNG_ERROR_BAD_NSEEDS	The value in NSeeds field is bad.
VSL_RNG_ERROR_BAD_STREAM	The random stream is invalid.
VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE	The value in StreamStateSize field is bad.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_RNG_ERROR_BAD_WORD_SIZE	The value in WordSize field is bad.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.

Status Code	Description
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	Two BRNGs are not compatible for the operation.
VSL_RNG_ERROR_FILE_CLOSE	Error in closing the file.
VSL_RNG_ERROR_FILE_OPEN	Error in opening the file.
VSL_RNG_ERROR_FILE_READ	Error in reading the file.
VSL_RNG_ERROR_FILE_WRITE	Error in writing the file.
VSL_RNG_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is not valid.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator is exceeded.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED	Non-deterministic random number generator is not supported on the CPU running the application.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number using non-deterministic random number generator exceeds threshold (see Section 7.2.1.12 <i>Non-deterministic</i> in [VS Notes] for more details)
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

VS RNG Usage Model

A typical algorithm for VS random number generators is as follows:

1. Create and initialize stream/streams. Functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`.
2. Call one or more RNGs.

3. Process the output.
4. Delete the stream/streams. Function `vslDeleteStream`.

NOTE

You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of a random stream that is output of basic generator MT19937. The seed is equal to 777. The stream is used to generate 10,000 normally distributed random numbers in blocks of 1,000 random numbers with parameters $a = 5$ and $\sigma = 2$. Delete the streams after completing the generation. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

Example of VS RNG Usage

```

include 'mkl_vsl.f90'

program MKL_VSL_GAUSSIAN

USE MKL_VSL_TYPE
USE MKL_VSL

real(kind=8) r(1000) ! buffer for random numbers
real(kind=8) s      ! average
real(kind=8) a, sigma ! parameters of normal distribution

TYPE (VSL_STREAM_STATE) :: stream

integer(kind=4) errcode
integer(kind=4) i,j
integer brng,method,seed,n

n = 1000
s = 0.0
a = 5.0
sigma = 2.0
brng=VSL_BRNG_MT19937
method=VSL_RNG_METHOD_GAUSSIAN_ICDF
seed=777

! ***** Initializing *****
errcode=vslnewstream( stream, brng, seed )

! ***** Generating *****
do i = 1,10
  errcode=vdrnggaussian( method, stream, n, r, a, sigma )
  do j = 1, 1000
    s = s + r(j)
  end do
end do

s = s / 10000.0

! ***** Deinitialize *****
errcode=vsldeletestream( stream )

! ***** Printing results *****

```

```
print *, "Sample mean of normal distribution = ", s
end
```

Additionally, examples that demonstrate usage of VS random number generators are available in:

`$(MKL)/examples/vslf/source`

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. [Table "Service Routines"](#) lists all available service routines

Service Routines

Routine	Short Description
<code>vslNewStream</code>	Creates and initializes a random stream.
<code>vslNewStreamEx</code>	Creates and initializes a random stream for the generators with multiple initial conditions.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for integer arrays.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for double precision floating-point arrays.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for single precision floating-point arrays.
<code>vslDeleteStream</code>	Deletes previously created stream.
<code>vslCopyStream</code>	Copies a stream to another stream.
<code>vslCopyStreamState</code>	Creates a copy of a random stream state.
<code>vslSaveStreamF</code>	Writes a stream to a binary file.
<code>vslLoadStreamF</code>	Reads a stream from a binary file.
<code>vslSaveStreamM</code>	Writes a random stream descriptive data, including state, to a memory buffer.
<code>vslLoadStreamM</code>	Creates a new stream and reads stream descriptive data, including state, from the memory buffer.
<code>vslGetStreamSize</code>	Computes size of memory necessary to hold the random stream.
<code>vslLeapfrogStream</code>	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
<code>vslSkipAheadStream</code>	Initializes the stream by the skip-ahead method.
<code>vslGetStreamStateBrng</code>	Obtains the index of the basic generator responsible for the generation of a given random stream.
<code>vslGetNumRegBrngs</code>	Obtains the number of currently registered basic generators.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (`vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`).
2. Generating random numbers with given distribution, see [Distribution Generators](#).

3. Deleting the stream ([vslDeleteStream](#)).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the [vslDeleteStream](#) function to delete all the streams afterwards.

vslNewStream

Creates and initializes a random stream.

Syntax

```
status = vslnewstream( stream, brng, seed )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>brng</i>	INTEGER, INTENT (IN)	Index of the basic generator to initialize the stream. See Table Values of <i>brng</i> parameter for specific value.
<i>seed</i>	INTEGER, INTENT (IN)	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Stream state descriptor

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. Use this function to create and initialize a new stream with a 32-bit seed only. If you need to provide multiple initial conditions such as several 32-bit or wider seeds, use the function [vslNewStreamEx](#). See [VS Notes](#) for a more detailed description of stream initialization for different basic generators.

NOTE

This function is not applicable for abstract basic random number generators. Please use [vslNewAbstractStream](#), [vslsNewAbstractStream](#) or [vslDNewAbstractStream](#) to utilize integer, single-precision or double-precision external random data respectively.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice

effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslNewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

```
status = vslnewstreamex( stream, brng, n, params )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>brng</i>	INTEGER, INTENT (IN)	Index of the basic generator to initialize the stream. See Table "Values of <i>brng</i> parameter" for specific value.
<i>n</i>	INTEGER, INTENT (IN)	Number of initial conditions contained in <i>params</i>
<i>params</i>	INTEGER (KIND=4), INTENT (IN)	Array of initial conditions necessary for the basic generator <i>brng</i> to initialize the stream. In the case of a quasi-random number generator only the first element in <i>params</i> parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Stream state descriptor

Description

The `vslNewStreamEx` function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use `vslNewStream`, which is analogous to `vslNewStreamEx` except that it takes only one 32-bit initial condition. In particular, `vslNewStreamEx` may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in [VS Notes](#).

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See [VS Notes](#) for the format for their passing and registration in VS.

NOTE

This function is not applicable for abstract basic random number generators. Please use `vslNewAbstractStream`, `vslsNewAbstractStream` or `vslldNewAbstractStream` to utilize integer, single-precision or double-precision external random data respectively.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

`vslNewAbstractStream`

Creates and initializes an abstract random stream for integer arrays.

Syntax

```
status = vslinewabstractstream( stream, n, ibuf, icallback )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Size of the array <i>ibuf</i>
<i>ibuf</i>	INTEGER (KIND=4), INTENT (IN)	Array of <i>n</i> 32-bit integers
<i>icallback</i>	See <i>Note</i> below	Address of the callback function used for <i>ibuf</i> update

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION IUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
```

```
INTEGER*4 stream(2)
```

```
INTEGER n
```

```
INTEGER*4 ibuf(n)
```

```
INTEGER nmin
```

```
INTEGER nmax
```

```
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION IUPDATEFUNC[C]( stream, n, ibuf, nmin, nmax, idx )
```

```
TYPE (VSL_STREAM_STATE), POINTER :: stream[reference]
```

```
INTEGER (KIND=4), INTENT (IN) :: n[reference]
```

```
INTEGER (KIND=4), INTENT (OUT) :: ibuf[reference] (0:n-1)
```

```
INTEGER (KIND=4), INTENT (IN) :: nmin[reference]
```

```
INTEGER (KIND=4), INTENT (IN) :: nmax[reference]
```

```
INTEGER (KIND=4), INTENT (IN) :: idx[reference]
```

The callback function returns the number of elements in the array actually updated by the function. [Table icallback Callback Function Parameters](#) gives the description of the callback function parameters.

icallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>ibuf</i>
<i>ibuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update

Parameters	Short Description
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>ibuf</i> to start update $0 \leq idx < n$.

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE(VSL_STREAM_STATE), TINTENT(OUT)	Descriptor of the stream state structure

Description

The `vslNewAbstractStream` function creates a new abstract stream and associates it with an integer array *ibuf* and your callback function *icallback* that is intended for updating of *ibuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

`vslNewAbstractStream`

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

```
status = vslnewabstractstream( stream, n, dbuf, a, b, dcallback )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT(IN)	Size of the array <i>dbuf</i>
<i>dbuf</i>	REAL(KIND=8), INTENT(IN)	Array of <i>n</i> double precision floating-point random numbers with uniform distribution over interval (a,b)
<i>a</i>	REAL(KIND=8), INTENT(IN)	Left boundary a
<i>b</i>	REAL(KIND=8), INTENT(IN)	Right boundary b
<i>dcallback</i>	See Note below	Address of the callback function used for update of the array <i>dbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Descriptor of the stream state structure

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION DUPDATEFUNC( stream, n, dbuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
DOUBLE PRECISION dbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION DUPDATEFUNC[C]( stream, n, dbuf, nmin, nmax, idx )
TYPE (VSL_STREAM_STATE), POINTER :: stream[reference]
INTEGER (KIND=4), INTENT (IN) :: n[reference]
REAL (KIND=8), INTENT (OUT) :: dbuf[reference] (0:n-1)
INTEGER (KIND=4), INTENT (IN) :: nmin[reference]
INTEGER (KIND=4), INTENT (IN) :: nmax[reference]
INTEGER (KIND=4), INTENT (IN) :: idx[reference]
```

The callback function returns the number of elements in the array actually updated by the function. [Table dcallback Callback Function Parameters](#) gives the description of the callback function parameters.

dcallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>dbuf</i>
<i>dbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>dbuf</i> to start update $0 \leq idx < n$.

Description

The `vslNewAbstractStream` function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a double precision array *dbuf* and your callback function *dcallback* that is intended for updating of *dbuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.

VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vslsNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

```
status = vslsnewabstractstream( stream, n, sbuf, a, b, scallback )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER, INTENT (IN)	Size of the array <i>sbuf</i>
<i>sbuf</i>	REAL (KIND=4), INTENT (IN)	Array of <i>n</i> single precision floating-point random numbers with uniform distribution over interval (a,b)
<i>a</i>	REAL (KIND=4), INTENT (IN)	Left boundary a
<i>b</i>	REAL (KIND=4), INTENT (IN)	Right boundary b
<i>scallback</i>	See <i>Note</i> below	Address of the callback function used for update of the array <i>sbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Descriptor of the stream state structure

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION SUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
REAL sbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION SUPDATEFUNC[C]( stream, n, sbuf, nmin, nmax, idx )
TYPE (VSL_STREAM_STATE), POINTER :: stream[reference]
INTEGER (KIND=4), INTENT (IN)    :: n[reference]
REAL (KIND=4), INTENT (OUT)     :: sbuf[reference] (0:n-1)
```

```

INTEGER(KIND=4), INTENT(IN)      :: nmin[reference]
INTEGER(KIND=4), INTENT(IN)      :: nmax[reference]
INTEGER(KIND=4), INTENT(IN)      :: idx[reference]

```

The callback function returns the number of elements in the array actually updated by the function. [Table scallback Callback Function Parameters](#) gives the description of the callback function parameters.

scallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>sbuf</i>
<i>sbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>sbuf</i> to start update $0 \leq idx < n$.

Description

The `vslsNewAbstractStream` function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a single precision array *sbuf* and your callback function *scallback* that is intended for updating of *sbuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vslDeleteStream

Deletes a random stream.

Syntax

```
status = vsldeletestream( stream )
```

Include Files

- `mk1.fi`, `mk1_vsl.f90`

Input/Output Parameters

Name	Type	Description
<i>stream</i>	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the descriptor becomes invalid.

Description

The function deletes the random stream created by one of the initialization functions.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

vslCopyStream

Creates a copy of a random stream.

Syntax

```
status = vslcopystream( newstream, srcstream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>srcstream</i>	TYPE(VSL_STREAM_STATE), INTENT(IN)	Descriptor of the stream to be copied

Output Parameters

Name	Type	Description
<i>newstream</i>	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Copied random stream descriptor

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>srcstream</i> parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>srcstream</i> is not a valid random stream.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>newstream</i> .

vslCopyStreamState

Creates a copy of a random stream state.

Syntax

```
status = vslcopystreamstate( deststream, srcstream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>srcstream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the destination stream where the state of <i>srcstream</i> stream is copied

Output Parameters

Name	Type	Description
<i>deststream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Descriptor of the stream with the state to be copied

Description

The `vslCopyStreamState` function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. An error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike `vslCopyStream` function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function `vslCopyStreamState` copies only *srcstream* stream state data to the generated *deststream* stream.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>srcstream</i> or <i>deststream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	Either <i>srcstream</i> or <i>deststream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	BRNG associated with <i>srcstream</i> is not compatible with BRNG associated with <i>deststream</i> .

vslSaveStreamF

Writes random stream descriptive data, including stream state, to binary file.

Syntax

```
errstatus = vslsavestreamf( stream, fname )
```

Include Files

- `mk1.fi`, `mk1_vsl.f90`

Input Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Random stream to be written to the file
<i>fname</i>	CHARACTER (*), INTENT (IN)	File name specified as a C-style null-terminated string

Output Parameters

Name	Type	Description
<i>errstatus</i>	INTEGER	Error status of the operation

Description

The `vslSaveStreamF` function writes the random stream descriptive data, including the stream state, to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. The random stream *stream* must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. The random stream can be read from the binary file using the `vslLoadStreamF` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>fname</i> or <i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

vslLoadStreamF

Creates new stream and reads stream descriptive data, including stream state, from binary file.

Syntax

```
errstatus = vslloadstreamf( stream, fname )
```

Include Files

- `mk1.fi`, `mk1_vsl.f90`

Input Parameters

Name	Type	Description
<i>fname</i>	CHARACTER (*), INTENT (IN)	File name specified as a C-style null-terminated string

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (OUT)	Descriptor of a new random stream
<i>errstatus</i>	INTEGER	Error status of the operation

Description

The `vslLoadStreamF` function creates a new stream and reads stream descriptive data, including the stream state, from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, an I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use `vslSaveStreamF` function.

CAUTION

Calling `vslLoadStreamF` with a previously initialized *stream* pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamF`, you should call the `vslDeleteStream` function first to deallocate the resources.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>fname</i> is a NULL pointer.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_FILE_FORMAT	Unknown file format.

VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is unsupported.
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslSaveStreamM

Writes random stream descriptive data, including stream state, to a memory buffer.

Syntax

```
errstatus = vslsavestreamm( stream, memptr )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Random stream to be written to the memory
<i>memptr</i>	INTEGER (KIND=1), DIMENSION (*), INTENT (IN)	Memory buffer to save random stream descriptive data to

Output Parameters

Name	Type	Description
<i>errstatus</i>	INTEGER	Error status of the operation

Description

The `vslSaveStreamM` function writes the random stream descriptive data, including the stream state, to the memory at *memptr*. Random stream *stream* must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. The *memptr* parameter must be a valid pointer to the memory of size sufficient to hold the random stream *stream*. Use the service routine `vslGetStreamSize` to determine this amount of memory.

If the stream cannot be saved to the memory, *errstatus* has a non-zero value. The random stream can be read from the memory pointed by *memptr* using the `vslLoadStreamM` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>memptr</i> or <i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is a NULL pointer.

vslLoadStreamM

Creates a new stream and reads stream descriptive data, including stream state, from the memory buffer.

Syntax

```
errstatus = vslloadstreamm( stream, memptr )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>memptr</i>	INTEGER(KIND=1), DIMENSION(*), INTENT(IN)	Memory buffer to load random stream descriptive data from

Output Parameters

Name	Type	Description
<i>stream</i>	TYPE(VSL_STREAM_STATE), INTENT(OUT)	Descriptor of a new random stream
<i>errstatus</i>	INTEGER	Error status of the operation

Description

The `vslLoadStreamM` function creates a new stream and reads stream descriptive data, including the stream state, from the memory buffer. A new random stream is created using the stream descriptive data from the memory pointer by `memptr`. If the stream cannot be read (for example, `memptr` is invalid), `errstatus` has a non-zero value. To save random stream to the memory, use `vslSaveStreamM` function. Use the service routine `vslGetStreamSize` to determine the amount of memory sufficient to hold the random stream.

CAUTION

Calling `LoadStreamM` with a previously initialized `stream` pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamM`, you should call the `vslDeleteStream` function first to deallocate the resources.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>memptr</i> is a NULL pointer.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_MEM_FORMAT	Descriptive random stream format is unknown.
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslGetStreamSize

Computes size of memory necessary to hold the random stream.

Syntax

```
memsize = vslgetstreamsize( stream )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Random stream

Output Parameters

Name	Type	Description
<i>memsize</i>	INTEGER	Amount of memory in bytes necessary to hold descriptive data of random stream <i>stream</i>

Description

The `vslGetStreamSize` function returns the size of memory in bytes which is necessary to hold the given random stream. Use the output of the function to allocate the buffer to which you will save the random stream by means of the `vslSaveStreamM` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is a NULL pointer.

vslLeapfrogStream

Initializes a stream using the leapfrog method.

Syntax

```
status = vsllLeapfrogStream( stream, k, nstreams )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

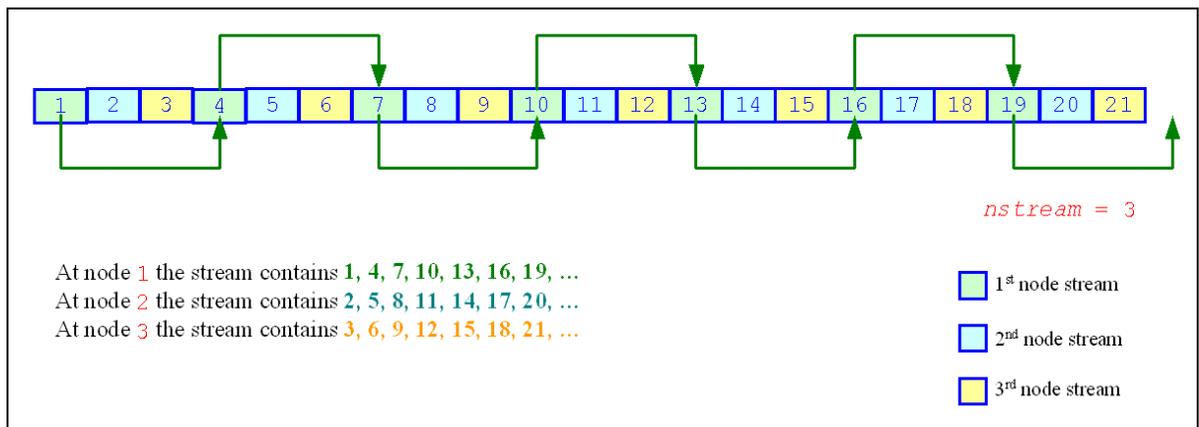
Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream to which leapfrog method is applied
<i>k</i>	INTEGER, INTENT (IN)	Index of the computational node, or stream number
<i>nstreams</i>	INTEGER, INTENT (IN)	Largest number of computational nodes, or stride

Description

The `vslLeapfrogStream` function generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the *nstreams* buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The function initializes the original random stream (see Figure "Leapfrog Method") to generate random numbers for the computational node *k*, $0 \leq k < nstreams$, where *nstreams* is the largest number of computational nodes used.

Leapfrog Method



The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See [VS Notes](#) for details.

For quasi-random basic generators, the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case *nstreams* parameter should be equal to the dimension of the quasi-random vector while *k* parameter should be the index of a component to be generated ($0 \leq k < nstreams$). Other parameters values are not allowed.

The following code illustrates the initialization of three independent streams using the leapfrog method:

Code for Leapfrog Method

```

...
TYPE (VSL_STREAM_STATE)    ::stream1
TYPE (VSL_STREAM_STATE)    ::stream2
TYPE (VSL_STREAM_STATE)    ::stream3

! Creating 3 identical streams
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
status = vslcopystream(stream2, stream1)
status = vslcopystream(stream3, stream1)

! Leapfrogging the streams
status = vslleapfrogstream(stream1, 0, 3)
status = vslleapfrogstream(stream2, 1, 3)
status = vslleapfrogstream(stream3, 2, 3)

! Generating random numbers
...
! Deleting the streams
status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...

```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

vslSkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

```
status = vslskipaheadstream( stream, nskip )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

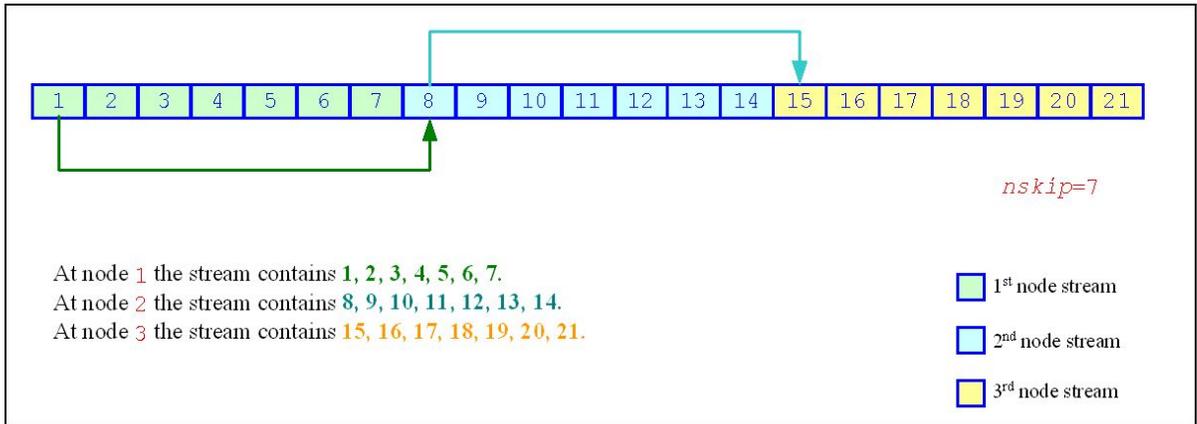
Name	Type	Description
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream to which block-splitting method is applied
<i>nskip</i>	INTEGER (KIND=8), INTENT (IN)	Number of skipped elements

Description

The `vslSkipAheadStream` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*,

then the original random sequence may be split by `vslSkipAheadStream` into non-overlapping blocks of `nskip` size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure "Block-Splitting Method").

Block-Splitting Method



The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See [VS Notes](#) for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip `NS` quasi-random vectors, set the `nskip` parameter equal to the `NS*DIMEN`, where `DIMEN` is the dimension of the quasi-random vector. If this operation results in exceeding the period of the quasi-random number generator, which is $2^{32}-1$, the library returns the `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED` error code.

The following code illustrates how to initialize three independent streams using the `vslSkipAheadStream` function:

Code for Block-Splitting Method

```
...
type (VSL_STREAM_STATE)  ::stream1
type (VSL_STREAM_STATE)  ::stream2
type (VSL_STREAM_STATE)  ::stream3

! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)

! Skipping ahead by 7 elements the 2nd stream
status = vslcopystream(stream2, stream1);
status = vslskipaheadstream(stream2, 7);

! Skipping ahead by 7 elements the 3rd stream
status = vslcopystream(stream3, stream2);
status = vslskipaheadstream(stream3, 7);

! Generating random numbers
...
! Deleting the streams
status = vsldeletestream(stream1)
```

```
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support the Skip-Ahead method.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the quasi-random number generator is exceeded.

vslGetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

```
brng = vslgetstreamstatebrng( stream )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>stream</i>	TYPE(VSL_STREAM_STATE), TINTENT(IN)	Descriptor of the stream state

Output Parameters

Name	Type	Description
<i>brng</i>	INTEGER	Index of the basic generator assigned for the generation of <i>stream</i> ; negative in case of an error

Description

The `vslGetStreamStateBrng` function retrieves the index of a basic generator used for generation of a given random stream.

Return Values

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

vslGetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

```
nregbrngs = vslgetnumregbrngs( )
```

Include Files

- mkl.fi, mkl_vsl.f90

Output Parameters

Name	Type	Description
<i>nregbrngs</i>	INTEGER	Number of basic generators registered at the moment of the function call

Description

The `vslGetNumRegBrngs` function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by the `VSL_MAX_REG_BRNGS` parameter.

Distribution Generators

Intel MKL VS routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. [Table "Continuous Distribution Generators"](#) and [Table "Discrete Distribution Generators"](#) list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	s, d	s, d	Uniform continuous distribution on the interval $[a,b]$
<code>vRngGaussian</code>	s, d	s, d	Normal (Gaussian) distribution
<code>vRngGaussianMV</code>	s, d	s, d	Multivariate normal (Gaussian) distribution
<code>vRngExponential</code>	s, d	s, d	Exponential distribution
<code>vRngLaplace</code>	s, d	s, d	Laplace distribution (double exponential distribution)
<code>vRngWeibull</code>	s, d	s, d	Weibull distribution
<code>vRngCauchy</code>	s, d	s, d	Cauchy distribution
<code>vRngRayleigh</code>	s, d	s, d	Rayleigh distribution
<code>vRngLognormal</code>	s, d	s, d	Lognormal distribution
<code>vRngGumbel</code>	s, d	s, d	Gumbel (extreme value) distribution
<code>vRngGamma</code>	s, d	s, d	Gamma distribution
<code>vRngBeta</code>	s, d	s, d	Beta distribution

Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	i	d	Uniform discrete distribution on the interval $[a,b)$
<code>vRngUniformBits</code>	i	i	Underlying BRNG integer recurrence
<code>vRngUniformBits 32</code>	i	i	Uniformly distributed bits in 32-bit chunks
<code>vRngUniformBits 64</code>	i	i	Uniformly distributed bits in 64-bit chunks
<code>vRngBernoulli</code>	i	s	Bernoulli distribution
<code>vRngGeometric</code>	i	s	Geometric distribution
<code>vRngBinomial</code>	i	d	Binomial distribution
<code>vRngHypergeometric</code>	i	d	Hypergeometric distribution
<code>vRngPoisson</code>	i	s (for <code>VSL_RNG_METHOD_POISSON_POISNORM</code>) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for <code>VSL_RNG_METHOD_POISSON_PTPE</code>)	Poisson distribution
<code>vRngPoissonV</code>	i	s	Poisson distribution with varying mean
<code>vRngNegBinomial</code>	i	d	Negative binomial distribution, or Pascal distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval $[a,b]$ belong to this interval irrespective of what a and b values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
<code>vRngUniform</code>	s, d
<code>vRngExponential</code>	s, d
<code>vRngWeibull</code>	s, d
<code>vRngRayleigh</code>	s, d

Type of Distribution	Data Types
<code>vRngLognormal</code>	s, d
<code>vRngGamma</code>	s, d
<code>vRngBeta</code>	s, d

See additional details about accurate and fast mode of random number generation in [VS Notes](#).

New method names

The current version of Intel MKL has a modified structure of VS RNG method names. (See [RNG Naming Conventions](#) for details.) The old names are kept for backward compatibility. The set correspondence between the new and legacy method names for VS random number generators.

Method Names for Continuous Distribution Generators

RNG	Legacy Method Name	New Method Name
<code>vRngUniform</code>	VSL_METHOD_SUNIFORM_STD, VSL_METHOD_DUNIFORM_STD, VSL_METHOD_SUNIFORM_STD_ACCURATE, VSL_METHOD_DUNIFORM_STD_ACCURATE	VSL_RNG_METHOD_UNIFORM_STD, VSL_RNG_METHOD_UNIFORM_STD_ACCURATE
<code>vRngGaussian</code>	VSL_METHOD_SGAUSSIAN_BOXMULLER, VSL_METHOD_SGAUSSIAN_BOXMULLER2, VSL_METHOD_SGAUSSIAN_ICDF, VSL_METHOD_DGAUSSIAN_BOXMULLER, VSL_METHOD_DGAUSSIAN_BOXMULLER2, VSL_METHOD_DGAUSSIAN_ICDF	VSL_RNG_METHOD_GAUSSIAN_BOXMULLER, VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, VSL_RNG_METHOD_GAUSSIAN_ICDF
<code>vRngGaussianMV</code>	VSL_METHOD_SGAUSSIANMV_BOXMULLER, VSL_METHOD_SGAUSSIANMV_BOXMULLER2, VSL_METHOD_SGAUSSIANMV_ICDF, VSL_METHOD_DGAUSSIANMV_BOXMULLER, VSL_METHOD_DGAUSSIANMV_BOXMULLER2, VSL_METHOD_DGAUSSIANMV_ICDF	VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER, , VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2, VSL_RNG_METHOD_GAUSSIANMV_ICDF
<code>vRngExponential</code>	VSL_METHOD_SEXPONENTIAL_ICDF, VSL_METHOD_DEXPONENTIAL_ICDF, VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE, VSL_METHOD_DEXPONENTIAL_ICDF_ACCURATE	VSL_RNG_METHOD_EXPONENTIAL_ICDF, VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE
<code>vRngLaplace</code>	VSL_METHOD_SLAPLACE_ICDF, VSL_METHOD_DLAPLACEL_ICDF	VSL_RNG_METHOD_LAPLACE_ICDF
<code>vRngWeibull</code>	VSL_METHOD_SWEIBULL_ICDF, VSL_METHOD_DWEIBULL_ICDF, VSL_METHOD_SWEIBULL_ICDF_ACCURATE, VSL_METHOD_DWEIBULL_ICDF_ACCURATE	VSL_RNG_METHOD_WEIBULL_ICDF, VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE
<code>vRngCauchy</code>	VSL_METHOD_SCAUCHY_ICDF, VSL_METHOD_DCAUCHY_ICDF	VSL_RNG_METHOD_CAUCHY_ICDF
<code>vRngRayleigh</code>	VSL_METHOD_SRAYLEIGH_ICDF, VSL_METHOD_DRAYLEIGH_ICDF, VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE, VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE	VSL_RNG_METHOD_RAYLEIGH_ICDF, VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE

RNG	Legacy Method Name	New Method Name
vRngLognormal	VSL_METHOD_SLOGNORMAL_BOXMULLER2, VSL_METHOD_DLOGNORMAL_BOXMULLER2, VSL_METHOD_SLOGNORMAL_BOXMULLER2_A CCURATE, VSL_METHOD_DLOGNORMAL_BOXMULLER2_A CCURATE	VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 , VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 _ACCURATE
	VSL_METHOD_SLOGNORMAL_ICDF, VSL_METHOD_DLOGNORMAL_ICDF, VSL_METHOD_SLOGNORMAL_ICDF_ACCURAT E, VSL_METHOD_DLOGNORMAL_ICDF_ACCURAT E	VSL_RNG_METHOD_LOGNORMAL_ICDF, VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCUR ATE
vRngGumbel	VSL_METHOD_SGUMBEL_ICDF, VSL_METHOD_DGUMBEL_ICDF	VSL_RNG_METHOD_GUMBEL_ICDF
vRngGamma	VSL_METHOD_SGAMMA_GNORM, VSL_METHOD_DGAMMA_GNORM, VSL_METHOD_SGAMMA_GNORM_ACCURATE, VSL_METHOD_DGAMMA_GNORM_ACCURATE	VSL_RNG_METHOD_GAMMA_GNORM, VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE
vRngBeta	VSL_METHOD_SBETA_CJA, VSL_METHOD_DBETA_CJA, VSL_METHOD_SBETA_CJA_ACCURATE, VSL_METHOD_DBETA_CJA_ACCURATE	VSL_RNG_METHOD_BETA_CJA, VSL_RNG_METHOD_BETA_CJA_ACCURATE

Method Names for Discrete Distribution Generators

RNG	Legacy Method Name	New Method Name
vRngUniform	VSL_METHOD_IUNIFORM_STD	VSL_RNG_METHOD_UNIFORM_STD
vRngUniformBits	VSL_METHOD_IUNIFORMBITS_STD	VSL_RNG_METHOD_UNIFORMBITS_STD
vRngBernoulli	VSL_METHOD_IBERNOULLI_ICDF	VSL_RNG_METHOD_BERNOULLI_ICDF
vRngGeometric	VSL_METHOD_IGEOMETRIC_ICDF	VSL_RNG_METHOD_GEOMETRIC_ICDF
vRngBinomial	VSL_METHOD_IBINOMIAL_BTPE	VSL_RNG_METHOD_BINOMIAL_BTPE
vRngHypergeometric	VSL_METHOD_IHYPERGEOMETRIC_H2PE	VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE
vRngPoisson	VSL_METHOD_IPOISSON_PTPE, VSL_METHOD_IPOISSON_POISNORM	VSL_RNG_METHOD_POISSON_PTPE, VSL_RNG_METHOD_POISSON_POISNORM
vRngPoissonV	VSL_METHOD_IPOISSONV_POISNORM	VSL_RNG_METHOD_POISSONV_POISNORM
vRngNegBinomial	VSL_METHOD_INEGBINOMIAL_NBAR	VSL_RNG_METHOD_NEGBINOMIAL_NBAR

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

vRngUniform

Generates random numbers with uniform distribution.

Syntax

```
status = vsrnguniform( method, stream, n, r, a, b )
```

```
status = vdrnguniform( method, stream, n, r, a, b )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method; the specific values are as follows: VSL_RNG_METHOD_UNIFORM_STD VSL_RNG_METHOD_UNIFORM_STD_ACCURATE Standard method.
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT (IN) for vsrnguniform REAL (KIND=8), INTENT (IN) for vdrnguniform	Left bound a
<i>b</i>	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT (IN) for vsrnguniform REAL (KIND=8), INTENT (IN) for vdrnguniform	Right bound b

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrnguniform REAL (KIND=4), INTENT (OUT) for vsrnguniform REAL (KIND=8), INTENT (OUT) for vdrnguniform	Vector of <i>n</i> random numbers uniformly distributed over the interval [<i>a</i> , <i>b</i>)

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in \mathbb{R}$; $a < b$.

The probability density function is given by:

$$f(x) = \frac{1}{b - a} \quad \text{for } a \leq x < b$$

The cumulative distribution function is as follows:

$$F(x) = \frac{x - a}{b - a} \quad \text{for } a \leq x < b$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEED ED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussian

Generates normally distributed random numbers.

Syntax

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

```
status = vdrnggaussian( method, stream, n, r, a, sigma )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAUSSIAN_BOXMULLER VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2 VSL_RNG_METHOD_GAUSSIAN_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT (IN) for vsrnggaussian REAL (KIND=8), INTENT (IN) for vdrnggaussian	Mean value a .
<i>sigma</i>	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT (IN) for vsrnggaussian REAL (KIND=8), INTENT (IN) for vdrnggaussian	Standard deviation σ .

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnggaussian REAL (KIND=4), INTENT (OUT) for vsrnggaussian REAL (KIND=8), INTENT (OUT) for vdrnggaussian	Vector of n normally distributed random numbers

Description

The `vRngGaussian` function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ , where

$$a, \sigma \in \mathbb{R}; \sigma > 0.$$

The probability density function is given by:

$$f_{a,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\sigma}(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty.$$

The cumulative distribution function $F_{a,\sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$$F_{a,\sigma}(x) = \Phi\left(\frac{x-a}{\sigma}\right)$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

```
status = vsrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

```
status = vdrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2 VSL_RNG_METHOD_GAUSSIANMV_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of <i>d</i> -dimensional vectors to be generated
<i>dimen</i>	Fortran 90: INTEGER, INTENT (IN)	Dimension <i>d</i> ($d \geq 1$) of output random vectors
<i>mstorage</i>	INTEGER, INTENT (IN)	Matrix storage scheme for upper triangular matrix T^T . The routine supports three matrix storage schemes:

Name	Type	Description
		<ul style="list-style-type: none"> VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the routine. VSL_MATRIX_STORAGE_PACKED— upper triangle elements of T^T are packed by rows into a one-dimensional array. VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T^T are passed.
a	DOUBLE PRECISION for vdrnggaussianmv REAL (KIND=4), INTENT (IN) for vsrnggaussianmv REAL (KIND=8), INTENT (IN) for vdrnggaussianmv	Mean vector a of dimension d
t	DOUBLE PRECISION for vdrnggaussianmv REAL (KIND=4), INTENT (IN) for vsrnggaussianmv REAL (KIND=8), INTENT (IN) for vdrnggaussianmv	Elements of the upper triangular matrix passed according to the matrix T^T storage scheme $mstorage$.

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrnggaussianmv REAL (KIND=4), INTENT (OUT) for vsrnggaussianmv REAL (KIND=8), INTENT (OUT) for vdrnggaussianmv	Array of n random vectors of dimension $dimen$

Description

The `vRngGaussianMV` function generates random numbers with d -variate normal (Gaussian) distribution with mean value a and variance-covariance matrix C , where $a \in R^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp(-1/2(x - a)^T C^{-1}(x - a)),$$

where $x \in R^d$.

Matrix C can be represented as $C = TT^T$, where T is a lower triangular matrix - Cholesky factor of C .

VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngExponential

Generates exponentially distributed random numbers.

Syntax

```
status = vsrngexponential( method, stream, n, r, a, beta )
status = vdrngexponential( method, stream, n, r, a, beta )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_EXPONENTIAL_ICDF VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrngexponential REAL (KIND=4), INTENT (IN) for vsrngexponential REAL (KIND=8), INTENT (IN) for vdrngexponential	Displacement <i>a</i>
<i>beta</i>	DOUBLE PRECISION for vdrngexponential REAL (KIND=4), INTENT (IN) for vsrngexponential REAL (KIND=8), INTENT (IN) for vdrngexponential	Scalefactor β .

Output Parameters

Name	Type	Description
r	DOUBLE PRECISION for vdrngexponential REAL (KIND=4), INTENT (OUT) for vsrngexponential REAL (KIND=8), INTENT (OUT) for vdrngexponential	Vector of n exponentially distributed random numbers

Description

The `vRngExponential` function generates random numbers with exponential distribution that has displacement a and scalefactor β , where $a, \beta \in R ; \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp (-(x - a)) / \beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp (-(x - a)) / \beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLaplace

Generates random numbers with Laplace distribution.

Syntax

```
status = vsrnglaplace( method, stream, n, r, a, beta )
```

```
status = vdrnglaplace( method, stream, n, r, a, beta )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_LAPLACE_ICDF Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrnglaplace REAL (KIND=4), INTENT (IN) for vsrnglaplace REAL (KIND=8), INTENT (IN) for vdrnglaplace	Mean value a
<i>beta</i>	DOUBLE PRECISION for vdrnglaplace REAL (KIND=4), INTENT (IN) for vsrnglaplace	Scalefactor β .

Name	Type	Description
------	------	-------------

	REAL (KIND=8), INTENT (IN) for vdrnglaplace	
--	--	--

Output Parameters

Name	Type	Description
------	------	-------------

r	DOUBLE PRECISION for vdrnglaplace REAL (KIND=4), INTENT (OUT) for vsrnglaplace REAL (KIND=8), INTENT (OUT) for vdrnglaplace	Vector of n Laplace distributed random numbers
-----	--	--

Description

The `vRngLaplace` function generates random numbers with Laplace distribution with mean value (or average) a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$. The scalefactor value determines the standard deviation as

|||||

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \\ 1 - \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x < a \end{cases}, -\infty < x < +\infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngWeibull*Generates Weibull distributed random numbers.***Syntax**

```
status = vsrngweibull( method, stream, n, r, alpha, a, beta )
```

```
status = vdrngweibull( method, stream, n, r, alpha, a, beta )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_WEIBULL_ICDF VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Name	Type	Description
<i>alpha</i>	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(IN) for vsrngweibull REAL(KIND=8), INTENT(IN) for vdrngweibull	Shape α .
<i>a</i>	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(IN) for vsrngweibull REAL(KIND=8), INTENT(IN) for vdrngweibull	Displacement a
<i>beta</i>	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(IN) for vsrngweibull REAL(KIND=8), INTENT(IN) for vdrngweibull	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrngweibull REAL(KIND=4), INTENT(OUT) for vsrngweibull REAL(KIND=8), INTENT(OUT) for vdrngweibull	Vector of n Weibull distributed random numbers

Description

The `vRngWeibull` function generates Weibull distributed random numbers with displacement a , scalefactor β , and shape α , where $\alpha, \beta, a \in \mathbb{R}$; $\alpha > 0, \beta > 0$.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x - a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngCauchy

Generates Cauchy distributed random values.

Syntax

```
status = vsrngcauchy( method, stream, n, r, a, beta )
```

```
status = vdrngcauchy( method, stream, n, r, a, beta )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_CAUCHY_ICDF Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrngcauchy REAL (KIND=4), INTENT (IN) for vsrngcauchy REAL (KIND=8), INTENT (IN) for vdrngcauchy	Displacement a .
<i>beta</i>	DOUBLE PRECISION for vdrngcauchy REAL (KIND=4), INTENT (IN) for vsrngcauchy REAL (KIND=8), INTENT (IN) for vdrngcauchy	Scale factor β .

Output Parameters

Name	Type	Description
<i>x</i>	DOUBLE PRECISION for vdrngcauchy REAL (KIND=4), INTENT (OUT) for vsrngcauchy REAL (KIND=8), INTENT (OUT) for vdrngcauchy	Vector of n Cauchy distributed random numbers

Description

The function generates Cauchy distributed random numbers with displacement a and scale factor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta \left(1 + \left(\frac{x-a}{\beta} \right)^2 \right)}, \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{\alpha,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x - a}{\beta}\right), -\infty < x < +\infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngRayleigh

Generates Rayleigh distributed random values.

Syntax

```
status = vsrngrayleigh( method, stream, n, r, a, beta )
status = vdrngrayleigh( method, stream, n, r, a, beta )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_RAYLEIGH_ICDF VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrngrayscaleigh REAL (KIND=4), INTENT (IN) for vsrngrayscaleigh REAL (KIND=8), INTENT (IN) for vdrngrayscaleigh	Displacement <i>a</i>
<i>beta</i>	DOUBLE PRECISION for vdrngrayscaleigh REAL (KIND=4), INTENT (IN) for vsrngrayscaleigh REAL (KIND=8), INTENT (IN) for vdrngrayscaleigh	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrngrayscaleigh REAL (KIND=4), INTENT (OUT) for vsrngrayscaleigh REAL (KIND=8), INTENT (OUT) for vdrngrayscaleigh	Vector of <i>n</i> Rayleigh distributed random numbers

Description

The `vRngRayleigh` function generates Rayleigh distributed random numbers with displacement *a* and scalefactor β , where $a, \beta \in R ; \beta > 0$.

The Rayleigh distribution is a special case of the [Weibull](#) distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLognormal

Generates lognormally distributed random numbers.

Syntax

`status = vsrnglognormal(method, stream, n, r, a, sigma, b, beta)`

`status = vdrnglognormal(method, stream, n, r, a, sigma, b, beta)`

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	<p>Generation method. The specific values are as follows:</p> <p>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2</p> <p>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE</p> <p>Box Muller 2 based method</p> <p>VSL_RNG_METHOD_LOGNORMAL_ICDF</p> <p>VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE</p> <p>Inverse cumulative distribution function based method</p>
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrnglognormal REAL (KIND=4), INTENT (IN) for vsrnglognormal REAL (KIND=8), INTENT (IN) for vdrnglognormal	Average <i>a</i> of the subject normal distribution
<i>sigma</i>	DOUBLE PRECISION for vdrnglognormal REAL (KIND=4), INTENT (IN) for vsrnglognormal REAL (KIND=8), INTENT (IN) for vdrnglognormal	Standard deviation σ of the subject normal distribution
<i>b</i>	DOUBLE PRECISION for vdrnglognormal REAL (KIND=4), INTENT (IN) for vsrnglognormal REAL (KIND=8), INTENT (IN) for vdrnglognormal	Displacement <i>b</i>
<i>beta</i>	DOUBLE PRECISION for vdrnglognormal	Scalefactor β .

Name	Type	Description
------	------	-------------

	REAL (KIND=4), INTENT (IN) for vsrnglognormal	
	REAL (KIND=8), INTENT (IN) for vdrnglognormal	

Output Parameters

Name	Type	Description
------	------	-------------

r	DOUBLE PRECISION for vdrnglognormal	Vector of n lognormally distributed random numbers
	REAL (KIND=4), INTENT (OUT) for vsrnglognormal	
	REAL (KIND=8), INTENT (OUT) for vdrnglognormal	

Description

The `vRngLognormal` function generates lognormally distributed random numbers with average of distribution a and standard deviation σ of subject normal distribution, displacement b , and scalefactor β , where $a, \sigma, b, \beta \in \mathbb{R}; \sigma > 0, \beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\frac{[\ln((x-b)/\beta) - a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\ln((x-b)/\beta) - a)/\sigma, & x > b \\ 0, & x \leq b \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGumbel

Generates Gumbel distributed random values.

Syntax

```
status = vsrnggumbel( method, stream, n, r, a, beta )
```

```
status = vdrnggumbel( method, stream, n, r, a, beta )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GUMBEL_ICDF Inverse cumulative distribution function method
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	DOUBLE PRECISION for vdrnggumbel REAL (KIND=4), INTENT (IN) for vsrnggumbel REAL (KIND=8), INTENT (IN) for vdrnggumbel	Displacement <i>a</i> .

Name	Type	Description
<i>beta</i>	DOUBLE PRECISION for vdrnggumbel REAL (KIND=4), INTENT (IN) for vsrnggumbel REAL (KIND=8), INTENT (IN) for vdrnggumbel	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrnggumbel REAL (KIND=4), INTENT (OUT) for vsrnggumbel REAL (KIND=8), INTENT (OUT) for vdrnggumbel	Vector of n random numbers with Gumbel distribution

Description

The `vRngGumbel` function generates Gumbel distributed random numbers with displacement a and scalefactor β , where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{ \frac{1}{\beta} \exp\left(\frac{x-a}{\beta}\right) \exp(-\exp((x-a)/\beta)) \right\}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGamma

Generates gamma distributed random values.

Syntax

```
status = vsrnggamma( method, stream, n, r, alpha, a, beta )
```

```
status = vdrnggamma( method, stream, n, r, alpha, a, beta )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAMMA_GNORM VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>alpha</i>	DOUBLE PRECISION for vdrnggamma	Shape α .

Name	Type	Description
	REAL (KIND=4), INTENT (IN) for vsrnggamma	
	REAL (KIND=8), INTENT (IN) for vdrnggamma	
<i>a</i>	DOUBLE PRECISION for vdrnggamma	Displacement <i>a</i> .
	REAL (KIND=4), INTENT (IN) for vsrnggamma	
	REAL (KIND=8), INTENT (IN) for vdrnggamma	
<i>beta</i>	DOUBLE PRECISION for vdrnggamma	Scalefactor β .
	REAL (KIND=4), INTENT (IN) for vsrnggamma	
	REAL (KIND=8), INTENT (IN) for vdrnggamma	

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrnggamma	Vector of <i>n</i> random numbers with gamma distribution
	REAL (KIND=4), INTENT (OUT) for vsrnggamma	
	REAL (KIND=8), INTENT (OUT) for vdrnggamma	

Description

The vRngGamma function generates random numbers with gamma distribution that has shape parameter α , displacement *a*, and scale parameter β , where α, β , and $a \in \mathbb{R}$; $\alpha > 0, \beta > 0$.

The probability density function is given by:

$$f_{\alpha,a,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} (x - a)^{\alpha-1} e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha, a, \beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^\alpha} (y - a)^{\alpha-1} e^{-(y-a)/\beta} dy, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBeta

Generates beta distributed random values.

Syntax

```
status = vsrngbeta( method, stream, n, r, p, q, a, beta )
status = vdrngbeta( method, stream, n, r, p, q, a, beta )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_BETA_CJA VSL_RNG_METHOD_BETA_CJA_ACCURATE See brief description of the method CJA in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>p</i>	DOUBLE PRECISION for vdrngbeta REAL (KIND=4), INTENT (IN) for vsrngbeta REAL (KIND=8), INTENT (IN) for vdrngbeta	Shape <i>p</i>
<i>q</i>	DOUBLE PRECISION for vdrngbeta REAL (KIND=4), INTENT (IN) for vsrngbeta REAL (KIND=8), INTENT (IN) for vdrngbeta	Shape <i>q</i>
<i>a</i>	DOUBLE PRECISION for vdrngbeta REAL (KIND=4), INTENT (IN) for vsrngbeta REAL (KIND=8), INTENT (IN) for vdrngbeta	Displacement <i>a</i> .
<i>beta</i>	DOUBLE PRECISION for vdrngbeta REAL (KIND=4), INTENT (IN) for vsrngbeta REAL (KIND=8), INTENT (IN) for vdrngbeta	Scale factor β .

Output Parameters

Name	Type	Description
<i>r</i>	DOUBLE PRECISION for vdrngbeta	Vector of <i>n</i> random numbers with beta distribution

Name	Type	Description
------	------	-------------

	REAL (KIND=4), INTENT (OUT) for vsrngbeta	
--	--	--

	REAL (KIND=8), INTENT (OUT) for vdrngbeta	
--	--	--

Description

The `vRngBeta` function generates random numbers with beta distribution that has shape parameters p and q , displacement a , and scale parameter β , where p, q, a , and $\beta \in \mathbb{R}$; $p > 0, q > 0, \beta > 0$.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p,q)\beta^{p+q-1}} (x-a)^{p-1} (\beta+a-x)^{q-1}, & a \leq x < a + \beta \\ 0, & x < a, x \geq a + \beta \end{cases}, -\infty < x < \infty,$$

where $B(p, q)$ is the complete beta function.

The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}} (y-a)^{p-1} (\beta+a-y)^{q-1} dy, & a \leq x < a + \beta \\ 1, & x \geq a + \beta \end{cases}, -\infty < x < \infty.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.

VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

vRngUniform

Generates random numbers uniformly distributed over the interval $[a, b)$.

Syntax

```
status = virnguniform( method, stream, n, r, a, b )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method; the specific value is as follows: VSL_RNG_METHOD_UNIFORM_STD Standard method. Currently there is only one method for this distribution generator.
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	INTEGER (KIND=4), INTENT (IN)	Left interval bound <i>a</i>
<i>b</i>	INTEGER (KIND=4), INTENT (IN)	Right interval bound <i>b</i>

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of <i>n</i> random numbers uniformly distributed over the interval $[a, b)$

Description

The vRngUniform function generates random numbers uniformly distributed over the interval $[a, b)$, where *a*, *b* are the left and right bounds of the interval respectively, and $a, b \in \mathbb{Z}; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \dots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \leq x < b, x \in R. \\ 1, & x \geq b \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits

Generates bits of underlying BRNG integer recurrence.

Syntax

```
status = virnguniformbits( method, stream, n, r )
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS_STD</code>
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>wordsize</i> of the structure <code>VSL_BRNG_PROPERTIES</code> . The total number of bits that are actually used to store the value are contained in the field <i>nbits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of <code>VSLBRngProperties</code> .

Description

The `vRngUniformBits` function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit LCG only 24 higher bits of an integer value can be considered random. See [VS Notes](#) for details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits32

Generates uniformly distributed bits in 32-bit chunks.

Syntax

```
status = virnguniformbits32( method, stream, n, r )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS32_STD</code>
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
r	INTEGER (KIND=4), INTENT (OUT)	Vector of n 32-bit random integer numbers with uniform bit distribution.

Description

The `vRngUniformBits32` function generates uniformly distributed bits in 32-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits32` is designed to ensure each bit in the 32-bit chunk is uniformly distributed. See [VS Notes](#) for details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits64

Generates uniformly distributed bits in 64-bit chunks.

Syntax

```
status = virnguniformbits64( method, stream, n, r )
```

Include Files

- `mkl.fi`, `mkl_vs1.f90`

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS64_STD</code>
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=8), INTENT (OUT)	Vector of <i>n</i> 64-bit random integer numbers with uniform bit distribution.

Description

The `vRngUniformBits64` function generates uniformly distributed bits in 64-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits64` is designed to ensure each bit in the 64-bit chunk is uniformly distributed. See [VS Notes](#) for details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BRNG_NOT_SUPPORTED</code>	BRNG is not supported by the function.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
<code>VSL_RNG_ERROR_ARS5_NOT_SUPPORTED</code>	ARS-5 random number generator is not supported on the CPU running the application.

vRngBernoulli*Generates Bernoulli distributed random values.***Syntax**

```
status = virngbernoulli( method, stream, n, r, p )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_BERNOULLI_ICDF Inverse cumulative distribution function method.
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>p</i>	REAL (KIND=8), INTENT (IN)	Success probability p of a trial

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of n Bernoulli distributed random values

Description

The `vRngBernoulli` function generates Bernoulli distributed random numbers with probability p of a single trial success, where

$$p \in \mathbb{R}; 0 \leq p \leq 1.$$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success p , and to 0 with probability $1 - p$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in \mathbb{R}. \\ 1, & x \geq 1 \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGeometric

Generates geometrically distributed random values.

Syntax

status = virnggeometric(*method*, *stream*, *n*, *r*, *p*)

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_GEOMETRIC_ICDF Inverse cumulative distribution function method.
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Name	Type	Description
p	REAL (KIND=8), INTENT (IN)	Success probability p of a trial

Output Parameters

Name	Type	Description
r	INTEGER (KIND=4), INTENT (OUT)	Vector of n geometrically distributed random values

Description

The `vRngGeometric` function generates geometrically distributed random numbers with probability p of a single trial success, where $p \in R; 0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p .

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \geq x \end{cases} \quad x \in R.$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.

VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBinomial

Generates binomially distributed random numbers.

Syntax

```
status = virngbinomial( method, stream, n, r, ntrial, p )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_BINOMIAL_BTPE See brief description of the BTPE method in Table "Values of <method> in method parameter" .
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>ntrial</i>	INTEGER (KIND=4), INTENT (IN)	Number of independent trials m
<i>p</i>	REAL (KIND=8), INTENT (IN)	Success probability p of a single trial

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of n binomially distributed random values

Description

The `vRngBinomial` function generates binomially distributed random numbers with number of independent Bernoulli trials m , and with probability p of a single trial success, where $p \in \mathbb{R}$; $0 \leq p \leq 1$, $m \in \mathbb{N}$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p .

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1 - p)^{m-k}, & 0 \leq x < m, x \in \mathbb{R} \\ 1, & x \geq m \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngHypergeometric

Generates hypergeometrically distributed random values.

Syntax

```
status = virnghypergeometric( method, stream, n, r, l, s, m )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE See brief description of the H2PE method in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>l</i>	INTEGER (KIND=4), INTENT (IN)	Lot size <i>l</i>
<i>s</i>	INTEGER (KIND=4), INTENT (IN)	Size of sampling without replacement <i>s</i>
<i>m</i>	INTEGER (KIND=4), INTENT (IN)	Number of marked elements <i>m</i>

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of <i>n</i> hypergeometrically distributed random values

Description

The `vRngHypergeometric` function generates hypergeometrically distributed random values with lot size *l*, size of sampling *s*, and number of marked elements in the lot *m*, where $l, m, s \in \mathbb{N} \cup \{0\}$; $l \geq \max(s, m)$.

Consider a lot of *l* elements comprising *m* "marked" and *l*-*m* "unmarked" elements. A trial sampling without replacement of exactly *s* elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of *s* elements contains exactly *k* marked elements.

The probability distribution is given by:)

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}$$

, $k \in \{\max(0, s + m - l), \dots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{1,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - 1) \\ \sum_{k=\max(0, s+m-1)}^{\lfloor x \rfloor} \frac{C_m^k C_1^{s-k}}{C_1^s}, & \max(0, s + m - 1) \leq x \leq \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoisson

Generates Poisson distributed random values.

Syntax

```
status = virngpoisson( method, stream, n, r, lambda )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific values are as follows: VSL_RNG_METHOD_POISSON_PTPE VSL_RNG_METHOD_POISSON_POISNORM See brief description of the PTPE and POISNORM methods in Table "Values of <method> in method parameter".
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>lambda</i>	REAL (KIND=8), INTENT (IN)	Distribution parameter λ .

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of <i>n</i> Poisson distributed random values

Description

The `vRng"Poisson"` function generates Poisson distributed random numbers with distribution parameter λ , where $\lambda \in R; \lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k \in \{0, 1, 2, \dots\}$$

$k \in \{0, 1, 2, \dots\}$.

The cumulative distribution function is as follows:

$$F_{\lambda}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoissonV

Generates Poisson distributed random values with varying mean.

Syntax

```
status = virngpoissonv( method, stream, n, r, lambda )
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is as follows: VSL_RNG_METHOD_POISSONV_POISNORM See brief description of the POISNORM method in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	Descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated

Name	Type	Description
<i>lambda</i>	REAL (KIND=8), INTENT (IN)	Array of n distribution parameters λ_i .

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of n Poisson distributed random values

Description

The `vRngPoissonV` function generates n Poisson distributed random numbers $x_i (i = 1, \dots, n)$ with distribution parameter λ_i , where $\lambda_i \in \mathbb{R}; \lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in \mathbb{R}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEED ED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngNegBinomial

Generates random numbers with negative binomial distribution.

Syntax

```
status = virngnegbinomial( method, stream, n, r, a, p )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>method</i>	INTEGER, INTENT (IN)	Generation method. The specific value is: <code>VSL_RNG_METHOD_NEGBINOMIAL_NBAR</code> See brief description of the <code>NBAR</code> method in Table "Values of <method> in method parameter"
<i>stream</i>	TYPE (VSL_STREAM_STATE), INTENT (IN)	descriptor of the stream state structure.
<i>n</i>	INTEGER, INTENT (IN)	Number of random values to be generated
<i>a</i>	REAL (KIND=8), INTENT (IN)	The first distribution parameter <i>a</i>
<i>p</i>	REAL (KIND=8), INTENT (IN)	The second distribution parameter <i>p</i>

Output Parameters

Name	Type	Description
<i>r</i>	INTEGER (KIND=4), INTENT (OUT)	Vector of <i>n</i> random values with negative binomial distribution.

Description

The `vRngNegBinomial` function generates random numbers with negative binomial distribution and distribution parameters *a* and *p*, where $p, a \in R; 0 < p < 1; a > 0$.

If the first distribution parameter $a \in \mathbb{N}$, this distribution is the same as Pascal distribution. If $a \in \mathbb{N}$, the distribution can be interpreted as the expected time of a -th success in a sequence of Bernoulli trials, when the probability of success is p .

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^k p^a (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^k p^a (1 - p)^k, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in \mathbb{R}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Advanced Service Routines

This section describes service routines for obtaining properties of the previously registered basic generators ([vslGetBrngProperties](#)). See [VS Notes](#) ("Basic Generators" section of VS Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

NOTE

The `vslRegisterBrng` function is provided in C for registering a user-defined basic generator, but it is not supported for Fortran. If you need to use a user-defined generator in Fortran, use an abstract basic random generator and abstract stream as described in the VS Notes.

Data types

The Advanced Service routines refer to a structure defining the properties of the basic generator.

This structure is described in Fortran 90 as follows:

```
TYPE VSL_BRNG_PROPERTIES
  INTEGER streamstatesize
  INTEGER nseeds
  INTEGER includeszero
  INTEGER wordsize
  INTEGER nbits
  INTEGER reserved(8)
END TYPE VSL_BRNG_PROPERTIES
```

The following table provides brief descriptions of the fields engaged in the above structure:

Field Descriptions

Field	Short Description
Fortran: <code>streamstatesize</code>	The size, in bytes, of the stream state structure for a given basic generator.
Fortran: <code>nseeds</code>	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
Fortran: <code>includeszero</code>	Flag value indicating whether the generator can produce a random 0.
Fortran: <code>wordsize</code>	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
Fortran: <code>nbits</code>	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, <code>wordsize/WordSize</code> is equal to 8 (number of bytes used to store the random value), while <code>nbits/NBits</code> contains the actual number of bits occupied by the value (in this example, 48).
Fortran: <code>reserved(8)</code>	Reserved for internal use.

NOTE

Using Advanced Service routines for defining generators is not supported for Fortran, but you can use the Fortran interface to get information about a previously-registered generator using [vslGetBrngProperties](#).

vslGetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

```
status = vslgetbrngproperties( brng, properties )
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>brng</i>	Fortran: INTEGER(KIND=4), INTENT(IN)	Number (index) of the registered basic generator; used for identification. See specific values in Table "Values of <i>brng</i> parameter" . Negative values indicate the registration error.

Output Parameters

Name	Type	Description
<i>properties</i>	Fortran: TYPE(VSL_BRNG_PROPERTIES), INTENT(OUT)	Pointer to the structure containing properties of the generator with number <i>brng</i>

Description

The `vslGetBrngProperties` function returns a structure with properties of a given basic generator.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.

Convolution and Correlation

Intel MKL VS provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision real and complex data.

For correct definition of implemented operations, see the [Mathematical Notation and Definitions](#) section.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision real and complex data
- Fourier algorithms for multi-dimensional single and double precision real and complex data
- Direct algorithms for one-dimensional single and double precision real and complex data
- Direct algorithms for multi-dimensional single and double precision real and complex data

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

SCONF, SCORF

SCOND, SCORD

SDCON, SDCOR

DDCON, DDCOR

SDDCON, SDDCOR.

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources:

`${MKL}/examples/vslf/essl/vsl_wrappers`

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers:

`${MKL}/examples/vslf/essl`

The convolution and correlation API provides interfaces for Fortran 90 and C/89 languages. You can use the Fortran 90 interface with programs written in Fortran.

Intel MKL provides the `mkl_vsl.f90` header file. All header files are in the directory

`${MKL}/include`

See more details about the Fortran header in [Random Number Generators](#) section of this chapter.

The convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All the Intel MKL VS convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Naming Conventions

The names of routines in the convolution and correlation API are written in lowercase (`vslsconvexec`), while the names of Fortran types and constants are written in uppercase. The names are not case-sensitive.

The names of routines have the following structure:

`vsl[datatype]{conv|corr}<base name>`

where

- `vsl` is a prefix indicating that the routine belongs to Intel® MKL Vector Statistics.

- `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be `s` (for single precision real type), `d` (for double precision real type), `c` (for single precision complex type), or `z` (for double precision complex type).
- `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.
- `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
FORTRAN 77: <code>INTEGER*4 task</code> (2)	Pointer to a task descriptor for convolution
Fortran 90: <code>TYPE (VSL_CONV_TASK)</code>	
FORTRAN 77: <code>INTEGER*4 task</code> (2)	Pointer to a task descriptor for correlation
Fortran 90: <code>TYPE (VSL_CORR_TASK)</code>	
FORTRAN 77: <code>REAL*4</code>	Input/output user real data in single precision
Fortran 90: <code>REAL (KIND=4)</code>	
FORTRAN 77: <code>REAL*8</code>	Input/output user real data in double precision
Fortran 90: <code>REAL (KIND=8)</code>	
FORTRAN 77: <code>COMPLEX*8</code>	Input/output user complex data in single precision
Fortran 90: <code>COMPLEX (KIND=4)</code>	
FORTRAN 77: <code>COMPLEX*16</code>	Input/output user complex data in double precision
Fortran 90: <code>COMPLEX (KIND=8)</code>	
FORTRAN 77: <code>INTEGER</code>	All other data
Fortran 90: <code>INTEGER</code>	

Generic integer type (without specifying the byte size) is used for all integer data.

NOTE

The actual size of the generic integer type is platform-dependent. Before you compile your application, set an appropriate byte size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® MKL User's Guide*.

Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel MKL convolution and correlation API.

Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
<i>job</i>	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation
<i>type</i>	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
<i>precision</i>	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays <i>x,y,z</i> .
<i>mode</i>	explicit	integer	None	Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See SetMode for the list of named constants for this parameter.
<i>method</i>	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given <i>mode</i> . Setting this parameter to "auto" means that software will choose the best available method.
<i>internal_precision</i>	optional	integer	Set equal to the value of <i>precision</i>	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See SetInternalPrecision for the list of named constants for this parameter.
<i>dims</i>	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays <i>x,y,z</i> . Can be in the range from 1 to 7.
<i>x,y</i>	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
<i>z</i>	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
<i>xshape, yshape, zshape</i>	explicit	integer arrays	None	Define shapes of the arrays <i>x, y, z</i> . See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	explicit	integer arrays	None	Define strides within arrays <i>x, y, z</i> , that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.
<i>start</i>	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array <i>z</i> . See SetStart and Data Allocation for more information.

Name	Category	Type	Default Value Label	Description
<i>decimation</i>	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array <i>z</i> . See SetDecimation and Data Allocation for more information.

Task Status and Error Reporting

The task status is an integer value, which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings.

An error can be caused by invalid parameter values, a system fault like a memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, the constructor assigns the `VSL_STATUS_OK` status to the task. When processing the task afterwards, other routines such as editors or executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation, which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor is terminated and the task keeps the same error code.

Normally, every convolution or correlation function (except `DeleteTask`) returns the status assigned to the task while performing the function operation.

The header files define symbolic names for the status codes. These names are defined as integer constants via the `PARAMETER` operators.

If there is no error, the `VSL_STATUS_OK` status is returned, which is defined as zero:

```
F90/F95:      INTEGER(KIND=4) VSL_STATUS_OK
              PARAMETER (VSL_STATUS_OK = 0)
```

In case of an error, a non-zero error code is returned, which indicates the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files.

Convolution/Correlation Status Codes

Status Code	Description
<code>VSL_CC_ERROR_NOT_IMPLEMENTED</code>	Requested functionality is not implemented.
<code>VSL_CC_ERROR_ALLOCATION_FAILURE</code>	Memory allocation failure.
<code>VSL_CC_ERROR_BAD_DESCRIPTOR</code>	Task descriptor is corrupted.
<code>VSL_CC_ERROR_SERVICE_FAILURE</code>	A service function has failed.
<code>VSL_CC_ERROR_EDIT_FAILURE</code>	Failure while editing the task.
<code>VSL_CC_ERROR_EDIT_PROHIBITED</code>	You cannot edit this parameter.
<code>VSL_CC_ERROR_COMMIT_FAILURE</code>	Task commitment has failed.
<code>VSL_CC_ERROR_COPY_FAILURE</code>	Failure while copying the task.
<code>VSL_CC_ERROR_DELETE_FAILURE</code>	Failure while deleting the task.

Status Code	Description
VSL_CC_ERROR_BAD_ARGUMENT	Bad argument or task parameter.
VSL_CC_ERROR_JOB	Bad parameter: <i>job</i> .
SL_CC_ERROR_KIND	Bad parameter: <i>kind</i> .
VSL_CC_ERROR_MODE	Bad parameter: <i>mode</i> .
VSL_CC_ERROR_METHOD	Bad parameter: <i>method</i> .
VSL_CC_ERROR_TYPE	Bad parameter: <i>type</i> .
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: <i>external_precision</i> .
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: <i>internal_precision</i> .
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: <i>dims</i> .
VSL_CC_ERROR_XSHAPE	Bad parameter: <i>xshape</i> .
VSL_CC_ERROR_YSHAPE	Bad parameter: <i>yshape</i> .
	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{max}$.
VSL_CC_ERROR_ZSHAPE	Bad parameter: <i>zshape</i> .
VSL_CC_ERROR_XSTRIDE	Bad parameter: <i>xstride</i> .
VSL_CC_ERROR_YSTRIDE	Bad parameter: <i>ystride</i> .
VSL_CC_ERROR_ZSTRIDE	Bad parameter: <i>zstride</i> .
VSL_CC_ERROR_X	Bad parameter: <i>x</i> .
VSL_CC_ERROR_Y	Bad parameter: <i>y</i> .
VSL_CC_ERROR_Z	Bad parameter: <i>z</i> .
VSL_CC_ERROR_START	Bad parameter: <i>start</i> .
VSL_CC_ERROR_DECIMATION	Bad parameter: <i>decimation</i> .
VSL_CC_ERROR_OTHER	Another error.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. No additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form constructors but also assign particular data to the first operand vector used in the convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Each constructor routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

The [Table "Task Constructors"](#) lists available task constructors:

Task Constructors

Routine	Description
vslConvNewTask/vslCorrNewTask	Creates a new convolution or correlation task descriptor for a multidimensional case.
vslConvNewTask1D/vslCorrNewTask1D	Creates a new convolution or correlation task descriptor for a one-dimensional case.
vslConvNewTaskX/vslCorrNewTaskX	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
vslConvNewTaskX1D/vslCorrNewTaskX1D	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

[vslConvNewTask/vslCorrNewTask](#)

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax

`status = vslsconvnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vsldconvnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vslcconvnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vslzconvnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vslscorrnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vsldcorrnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vslccorrnewtask(task, mode, dims, xshape, yshape, zshape)`

`status = vslzcorrnewtask(task, mode, dims, xshape, yshape, zshape)`

Include Files

- `mkl.fi`, `mkl_vs1.f90`

Input Parameters

Name	Type	Description
<code>mode</code>	FORTRAN 77 : INTEGER Fortran 90 : INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.

Name	Type	Description
<i>dims</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*)	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*)	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*)	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtask, vsldconvnewtask, vsllconvnewtask, vslzconvnewtask INTEGER*4 task(2) for vsllcorrnewtask, vsldcorrnewtask, vsllcorrnewtask, vsllcorrnewtask Fortran 90: TYPE (VSL_CONV_TASK) for vslsconvnewtask, vsldconvnewtask, vsllconvnewtask, vslzconvnewtask TYPE (VSL_CORR_TASK) for vsllcorrnewtask, vsldcorrnewtask, vsllcorrnewtask, vsllcorrnewtask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTask/vslCorrNewTask` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

The parameters `xshape`, `yshape`, and `zshape` define the shapes of the input and output data provided by the arrays `x`, `y`, and `z`, respectively. Each shape parameter is an array of integers with its length equal to the value of `dims`. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter `dims` is 1, then `xshape`, `yshape`, `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. Note that values of shape parameters may differ from physical shapes of arrays `x`, `y`, and `z` if non-trivial strides are assigned.

If the constructor fails to create a task descriptor, it returns a `NULL` task pointer.

`vslConvNewTask1D/vslCorrNewTask1D`

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

```
status = vslsconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslcconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslzconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslscorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldcorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vslccorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vslzcorrnewtask1d(task, mode, xshape, yshape, zshape)
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>mode</code>	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<code>xshape</code>	INTEGER	Defines the length of the input data sequence for the source array <code>x</code> . See Data Allocation for more information.
<code>yshape</code>	INTEGER	Defines the length of the input data sequence for the source array <code>y</code> . See Data Allocation for more information.
<code>zshape</code>	INTEGER	Defines the length of the output data sequence to be stored in array <code>z</code> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	<p>FORTRAN 77: INTEGER*4 <code>task(2)</code> for <code>vslsconvnewtask1d</code>, <code>vsldconvnewtask1d</code>, <code>vslcconvnewtask1d</code>, <code>vslzconvnewtask1d</code></p> <p>INTEGER*4 <code>task(2)</code> for <code>vslscorrnewtask1d</code>, <code>vsldcorrnewtask1d</code>, <code>vslccorrnewtask1d</code>, <code>vslzcorrnewtask1d</code></p> <p>TYPE (VSL_CONV_TASK) for <code>vslsconvnewtask1d</code>, <code>vsldconvnewtask1d</code>, <code>vslcconvnewtask1d</code>, <code>vslzconvnewtask1d</code></p> <p>TYPE (VSL_CORR_TASK) for <code>vslscorrnewtask1d</code>, <code>vsldcorrnewtask1d</code>, <code>vslccorrnewtask1d</code>, <code>vslzcorrnewtask1d</code></p> <p>VSLCorrTaskPtr* for <code>vslsCorrNewTask1D</code>, <code>vsldCorrNewTask1D</code>, <code>vslcCorrNewTask1D</code>, <code>vslzCorrNewTask1D</code></p>	<p>Pointer to the task descriptor if created successfully or NULL pointer otherwise.</p>
<i>status</i>	INTEGER	<p>Set to <code>VSL_STATUS_OK</code> if the task is created successfully or set to non-zero error code otherwise.</p>

Description

Each `vslConvNewTask1D/vslCorrNewTask1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)). Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter `dims` is 1. The parameters `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. You explicitly assign the shape parameters when calling the constructor.

`vslConvNewTaskX/vslCorrNewTaskX`

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

```
status = vslsconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

```
status = vsldconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

```

status = vslcconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslccorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)

```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>mode</i>	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<i>dims</i>	INTEGER	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	INTEGER, DIMENSION(*)	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	INTEGER, DIMENSION(*)	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.
<i>x</i>	<p>REAL*8 for real data in double precision flavors,</p> <p>COMPLEX*8 for complex data in single precision flavors,</p> <p>COMPLEX*16 for complex data in double precision flavors</p> <p>REAL(KIND=4), DIMENSION (*) for real data in single precision flavors,</p> <p>REAL(KIND=8), DIMENSION (*) for real data in double precision flavors,</p> <p>COMPLEX(KIND=4), DIMENSION (*) for complex data in single precision flavors,</p> <p>COMPLEX(KIND=8), DIMENSION (*) for complex data in double precision flavors</p>	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.

Name	Type	Description
<i>xstride</i>	INTEGER, DIMENSION(*)	Strides for input data in the array <i>x</i> .

Output Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for <i>vslscorrnewtaskx</i> , <i>vsldcorrnewtaskx</i> , <i>vslccorrnewtaskx</i> , <i>vslzcorrnewtaskx</i> TYPE (VSL_CONV_TASK) for <i>vslsconvnewtaskx</i> , <i>vsldconvnewtaskx</i> , <i>vslcconvnewtaskx</i> , <i>vslzconvnewtaskx</i> TYPE (VSL_CORR_TASK) for <i>vslscorrnewtaskx</i> , <i>vsldcorrnewtaskx</i> , <i>vslccorrnewtaskx</i> , <i>vslzcorrnewtaskx</i> VSLCorrTaskPtr* for <i>vslsCorrNewTaskX</i> , <i>vsldCorrNewTaskX</i> , <i>vslcCorrNewTaskX</i> , <i>vslzCorrNewTaskX</i>	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	INTEGER	Set to <code>VSL_STATUS_OK</code> if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX/vslCorrNewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX/vslCorrNewTaskX` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see [vslConvDeleteTask/vslCorrDeleteTask](#)).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the

parameter *dims* is 1, then *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

The stride parameter *xstride* specifies the physical location of the input data in the array *x*. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is *s*, then only every *s*th element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

vslConvNewTaskX1D/vslCorrNewTaskX1D

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

```
status = vslsconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslldconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslcconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslzconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslldcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslccorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslzcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

Include Files

- mkl.fi, mkl_vs1.f90

Input Parameters

Name	Type	Description
<i>mode</i>	INTEGER	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<i>xshape</i>	INTEGER	Defines the length of the input data sequence for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	INTEGER	Defines the length of the input data sequence for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	INTEGER	Defines the length of the output data sequence to be stored in array <i>z</i> . See Data Allocation for more information.
<i>x</i>	REAL*8 for real data in double precision flavors, COMPLEX*8 for complex data in single precision flavors, COMPLEX*16 for complex data in double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.

Name	Type	Description
	REAL (KIND=4), DIMENSION (*) for real data in single precision flavors,	
	REAL (KIND=8), DIMENSION (*) for real data in double precision flavors,	
	COMPLEX (KIND=4), DIMENSION (*) for complex data in single precision flavors,	
	COMPLEX (KIND=8), DIMENSION (*) for complex data in double precision flavors	
<i>xstride</i>	INTEGER	Stride for input data sequence in the arrayx.

Output Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for vslscorrnewtaskx1d, vsldcorrnewtaskx1d, vslccorrnewtaskx1d, vslzcorrnewtaskx1d TYPE (VSL_CONV_TASK) for vslsconvnewtaskx1d, vsldconvnewtaskx1d, vslcconvnewtaskx1d, vslzconvnewtaskx1d TYPE (VSL_CORR_TASK) for vslscorrnewtaskx1d, vsldcorrnewtaskx1d, vslccorrnewtaskx1d, vslzcorrnewtaskx1d VSLCorrTaskPtr* for vslsCorrNewTaskX1D, vsldCorrNewTaskX1D, vslcCorrNewTaskX1D, vslzCorrNewTaskX1D	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	INTEGER	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter *dims* is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

The **stride parameters** *xstride* specifies the physical location of the input data in the array *x* and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is *s*, then only every *s*th element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

Task Editors

Task editors in convolution and correlation API of Intel MKL are routines intended for setting up or changing the following task parameters (see [Table "Convolution and Correlation Task Parameters"](#)):

- *mode*
- *internal_precision*
- *start*
- *decimation*

For setting up or changing each of the above parameters, a separate routine exists.

NOTE

Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

The work data computed during the last commitment process may become invalid with respect to new parameter settings. That is why after applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and goes through the full commitment process again during the next execution or copy operation. For more information on task commitment, see the [Introduction to Convolution and Correlation](#).

[Table "Task Editors"](#) lists available task editors.

Task Editors

Routine	Description
<code>vslConvSetMode/vslCorrSetMode</code>	Changes the value of the parameter <i>mode</i> for the operation of convolution or correlation.
<code>vslConvSetInternalPrecision/vslCorrSetInternalPrecision</code>	Changes the value of the parameter <i>internal_precision</i> for the operation of convolution or correlation.
<code>vslConvSetStart/vslCorrSetStart</code>	Sets the value of the parameter <i>start</i> for the operation of convolution or correlation.
<code>vslConvSetDecimation/vslCorrSetDecimation</code>	Sets the value of the parameter <i>decimation</i> for the operation of convolution or correlation.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslConvSetMode/vslCorrSetMode

Changes the value of the parameter `mode` in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetmode(task, newmode)
```

```
status = vslcorrsetmode(task, newmode)
```

Include Files

- `mk1.fi`, `mk1_vs1.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<p>FORTRAN 77: INTEGER*4 <code>task(2)</code> for <code>vslconvsetmode</code></p> <p>INTEGER*4 <code>task(2)</code> for <code>vslcorrsetmode</code></p> <p>Fortran 90: TYPE(VSL_CONV_TASK) for <code>vslconvsetmode</code></p> <p>TYPE(VSL_CORR_TASK) for <code>vslcorrsetmode</code></p>	Pointer to the task descriptor.
<code>newmode</code>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER</p>	New value of the parameter <code>mode</code> .

Output Parameters

Name	Type	Description
<code>status</code>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER</p>	Current status of the task.

Description

This function is declared in `mk1_vs1.f90` for the Fortran interface.

The function routine changes the value of the parameter `mode` for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for `mode` is assigned by a task constructor.

Predefined values for the `mode` parameter are as follows:

Values of *mode* parameter

Value	Purpose
VSL_CONV_MODE_FFT	Compute convolution by using fast Fourier transform.
VSL_CORR_MODE_FFT	Compute correlation by using fast Fourier transform.
VSL_CONV_MODE_DIRECT	Compute convolution directly.
VSL_CORR_MODE_DIRECT	Compute correlation directly.
VSL_CONV_MODE_AUTO	Automatically choose direct or Fourier mode for convolution.
VSL_CORR_MODE_AUTO	Automatically choose direct or Fourier mode for correlation.

vslConvSetInternalPrecision/vslCorrSetInternalPrecision

Changes the value of the parameter `internal_precision` in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetinternalprecision(task, precision)
status = vslcorrsetinternalprecision(task, precision)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for <code>vslcorrsetinternalprecision</code> TYPE(VSL_CONV_TASK) for <code>vslconvsetinternalprecision</code> TYPE(VSL_CORR_TASK) for <code>vslcorrsetinternalprecision</code>	Pointer to the task descriptor.
<i>precision</i>	INTEGER	New value of the parameter <code>internal_precision</code> .

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Current status of the task.

Description

The `vslConvSetInternalPrecision/vslCorrSetInternalPrecision` routine changes the value of the parameter `internal_precision` for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for `internal_precision` is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the *internal_precision* can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the *internal_precision* input parameter are as follows:

Values of *internal_precision* Parameter

Value	Purpose
VSL_CONV_PRECISION_SINGLE	Compute convolution with single precision.
VSL_CORR_PRECISION_SINGLE	Compute correlation with single precision.
VSL_CONV_PRECISION_DOUBLE	Compute convolution with double precision.
VSL_CORR_PRECISION_DOUBLE	Compute correlation with double precision.

vslConvSetStart/vslCorrSetStart

Changes the value of the parameter start in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetstart(task, start)
```

```
status = vslcorrsetstart(task, start)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for vslcorrsetstart TYPE(VSL_CONV_TASK) for vslconvsetstart TYPE(VSL_CORR_TASK) for vslcorrsetstart	Pointer to the task descriptor.
<i>start</i>	INTEGER, DIMENSION (*)	New value of the parameter <i>start</i> .

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Current status of the task.

Description

The vslConvSetStart/vslCorrSetStart routine sets the value of the parameter *start* for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, *start* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *start* is undefined and this parameter is not used. Therefore the only way to set and use the *start* parameter is via assigning it some value by one of the `vslConvSetStart/vslCorrSetStart` routines.

vslConvSetDecimation/vslCorrSetDecimation

Changes the value of the parameter decimation in the convolution or correlation task descriptor.

Syntax

```
status = vslconvsetdecimation(task, decimation)
```

```
status = vslcorrsetdecimation(task, decimation)
```

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for <code>vslcorrsetdecimation</code> TYPE(VSL_CONV_TASK) for <code>vslconvsetdecimation</code> TYPE(VSL_CORR_TASK) for <code>vslcorrsetdecimation</code>	Pointer to the task descriptor.
<i>decimatio</i> <i>n</i>	INTEGER, DIMENSION (*)	New value of the parameter <i>decimation</i> .

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Current status of the task.

Description

The routine sets the value of the parameter *decimation* for the operation of convolution or correlation. This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if *decimation* = *d* > 1, only every *d*-th element of the mathematical result is written to the output array *z*. In a multidimensional case, *decimation* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *decimation* is undefined and this parameter is not used. Therefore the only way to set and use the *decimation* parameter is via assigning it some value by one of the `vslSetDecimation` routines.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the user data supplied for input vectors.

After you create and adjust a task, you can execute it multiple times by applying to different input/output data of the same type, precision, and shape.

Intel MKL provides the following forms of convolution/correlation execution routines:

- **General form** executors that use the task descriptor created by the general form constructor and expect to get two source data arrays x and y on input
- **X-form** executors that use the task descriptor created by the X-form constructor and expect to get only one source data array y on input because the first array x has been already specified on the construction stage

When the task is executed for the first time, the execution routine includes a task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

Each execution routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

You can use the `NULL` task pointer in calls to execution routines. In this case, the routine is terminated and no system crash occurs.

If the task is executed successfully, the execution routine returns the zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`
- if the task descriptor is corrupted
- if calculation has failed for some other reason.

NOTE

Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Task Execution Routines

Routine	Description
<code>vslConvExec/vslCorrExec</code>	Computes convolution or correlation for a multidimensional case.
<code>vslConvExec1D/vslCorrExec1D</code>	Computes convolution or correlation for a one-dimensional case.
<code>vslConvExecX/vslCorrExecX</code>	Computes convolution or correlation as X-form for a multidimensional case.
<code>vslConvExecX1D/vslCorrExecX1D</code>	Computes convolution or correlation as X-form for a one-dimensional case.

`vslConvExec/vslCorrExec`

Computes convolution or correlation for multidimensional case.

Syntax

```
status = vslsconvexec(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslldconvexec(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslcconvexec(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslzconvexec(task, x, xstride, y, ystride, z, zstride)
```

`status = vslscorrexec(task, x, xstride, y, ystride, z, zstride)`

`status = vsldcorrexec(task, x, xstride, y, ystride, z, zstride)`

`status = vslccorrexec(task, x, xstride, y, ystride, z, zstride)`

`status = vslzcorrexec(task, x, xstride, y, ystride, z, zstride)`

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<p>INTEGER*4 <code>task(2)</code> for <code>vslscorrexec</code>, <code>vsldcorrexec</code>, <code>vslccorrexec</code>, <code>vslzcorrexec</code></p> <p>TYPE (VSL_CONV_TASK) for <code>vslsconvexec</code>, <code>vsldconvexec</code>, <code>vslcconvexec</code>, <code>vslzconvexec</code></p> <p>TYPE (VSL_CORR_TASK) for <code>vslscorrexec</code>, <code>vsldcorrexec</code>, <code>vslccorrexec</code>, <code>vslzcorrexec</code></p> <p>VSLCorrTaskPtr for <code>vslsCorrExec</code>, <code>vsldCorrExec</code>, <code>vslcCorrExec</code>, <code>vslzCorrExec</code></p>	Pointer to the task descriptor
<code>x, y</code>	<p>REAL*8 for <code>vsldconvexec</code> and <code>vsldcorrexec</code>,</p> <p>COMPLEX*8 for <code>vslcconvexec</code> and <code>vslccorrexec</code>,</p> <p>COMPLEX*16 for <code>vslzconvexec</code> and <code>vslzcorrexec</code></p> <p>REAL (KIND=4), DIMENSION (*) for <code>vslsconvexec</code> and <code>vslscorrexec</code>,</p> <p>REAL (KIND=8), DIMENSION (*) for <code>vsldconvexec</code> and <code>vsldcorrexec</code>,</p> <p>COMPLEX (KIND=4), DIMENSION (*) for <code>vslcconvexec</code> and <code>vslccorrexec</code>,</p> <p>COMPLEX (KIND=8), DIMENSION (*) for <code>vslzconvexec</code> and <code>vslzcorrexec</code></p>	Pointers to arrays containing input data. See Data Allocation for more information.
<code>xstride</code> , <code>ystride</code> , <code>zstride</code>	INTEGER, DIMENSION (*)	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	REAL*8 for <code>vslldconvexec</code> and <code>vslldcorrexec</code> , COMPLEX*8 for <code>vslcconvexec</code> and <code>vslccorrexec</code> , COMPLEX*16 for <code>vslzconvexec</code> and <code>vslzcorrexec</code> REAL(KIND=4), DIMENSION(*) for <code>vslsconvexec</code> and <code>vslscorrexec</code> , REAL(KIND=8), DIMENSION(*) for <code>vslldconvexec</code> and <code>vslldcorrexec</code> , COMPLEX(KIND=4), DIMENSION(*) for <code>vslcconvexec</code> and <code>vslccorrexec</code> , COMPLEX(KIND=8), DIMENSION(*) for <code>vslzconvexec</code> and <code>vslzcorrexec</code>	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	INTEGER	Set to <code>VSL_STATUS_OK</code> if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec/vslCorrExec` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask/vslCorrNewTask` constructor and pointed to by *task*. If *task* is NULL, no operation is done.

The stride parameters *xstride*, *ystride*, and *zstride* specify the physical location of the input and output data in the arrays *x*, *y*, and *z*, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *zstride* is *s*, then only every *s*th element of the array *z* will be used to store the output data. The stride value must be positive or negative but not zero.

`vslConvExec1D/vslCorrExec1D`

Computes convolution or correlation for one-dimensional case.

Syntax

```
status = vslsconvexecl(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslldconvexecl(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslcconvexecl(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslzconvexecl(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslscorrexecld(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslldcorrexecld(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslccorrexec1d(task, x, xstride, y, ystride, z, zstride)
```

```
status = vslzcorrexec1d(task, x, xstride, y, ystride, z, zstride)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	<p>INTEGER*4 <i>task</i>(2) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d</p> <p>TYPE(VSL_CONV_TASK) for vslsconvexec1d, vsldconvexec1d, vslcconvexec1d, vslzconvexec1d</p> <p>TYPE(VSL_CORR_TASK) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d</p> <p>VSLCorrTaskPtr for vslsCorrExec1D, vsldCorrExec1D, vslcCorrExec1D, vslzCorrExec1D</p>	Pointer to the task descriptor.
<i>x, y</i>	<p>REAL*8 for vsldconvexec1d and vsldcorrexec1d,</p> <p>COMPLEX*8 for vslcconvexec1d and vslccorrexec1d,</p> <p>COMPLEX*16 for vslzconvexec1d and vslzcorrexec1d</p> <p>REAL(KIND=4), DIMENSION(*) for vslsconvexec1d and vslscorrexec1d,</p> <p>REAL(KIND=8), DIMENSION(*) for vsldconvexec1d and vsldcorrexec1d,</p> <p>COMPLEX(KIND=4), DIMENSION (*) for vslcconvexec1d and vslccorrexec1d,</p>	Pointers to arrays containing input data. See Data Allocation for more information.

Name	Type	Description
	COMPLEX(KIND=8), DIMENSION (*) for <code>vslzconvexec1d</code> and <code>vslzcorrexec1d</code>	
<code>xstride</code> , <code>ystride</code> , <code>zstride</code>	INTEGER	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<code>z</code>	REAL*8 for <code>vslldconvexec1d</code> and <code>vslldcorrexec1d</code> , COMPLEX*8 for <code>vslcconvexec1d</code> and <code>vslccorrexec1d</code> , COMPLEX*16 for <code>vslzconvexec1d</code> and <code>vslzcorrexec1d</code> REAL(KIND=4), DIMENSION(*) for <code>vslsconvexec1d</code> and <code>vslscorrexec1d</code> , REAL(KIND=8), DIMENSION(*) for <code>vslldconvexec1d</code> and <code>vslldcorrexec1d</code> , COMPLEX(KIND=4), DIMENSION(*) for <code>vslcconvexec1d</code> and <code>vslccorrexec1d</code> , COMPLEX(KIND=8), DIMENSION(*) for <code>vslzconvexec1d</code> and <code>vslzcorrexec1d</code>	Pointer to the array that stores output data. See Data Allocation for more information.
<code>status</code>	INTEGER	Set to <code>VSL_STATUS_OK</code> if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec1D/vslCorrExec1D` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter `dims` is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask1D/vslCorrNewTask1D` constructor and pointed to by `task`. If `task` is NULL, no operation is done.

`vslConvExecX/vslCorrExecX`

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

```

status = vslsconvexecx(task, y, ystride, z, zstride)
status = vsldconvexecx(task, y, ystride, z, zstride)
status = vslcconvexecx(task, y, ystride, z, zstride)
status = vslzconvexecx(task, y, ystride, z, zstride)
status = vslscorexecx(task, y, ystride, z, zstride)
status = vsldcorexecx(task, y, ystride, z, zstride)
status = vslccorexecx(task, y, ystride, z, zstride)
status = vslzcorexecx(task, y, ystride, z, zstride)

```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for vslscorexecx, vsldcorexecx, vslccorexecx, vslzcorexecx TYPE (VSL_CONV_TASK) for vslsconvexecx, vsldconvexecx, vslcconvexecx, vslzconvexecx TYPE (VSL_CORR_TASK) for vslscorexecx, vsldcorexecx, vslccorexecx, vslzcorexecx VSLCorrTaskPtr for vslsCorrExecX, vsldCorrExecX, vslcCorrExecX, vslzCorrExecX	Pointer to the task descriptor.
<i>x, y</i>	REAL*8 for vsldconvexecx and vsldcorexecx, COMPLEX*8 for vslcconvexecx and vslccorexecx, COMPLEX*16 for vslzconvexecx and vslzcorexecx REAL (KIND=4), DIMENSION (*) for vslsconvexecx and vslscorexecx,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	REAL(KIND=8), DIMENSION(*) for <code>vsldconvexecx</code> and <code>vsldcorreexecx</code> ,	
	COMPLEX(KIND=4), DIMENSION (*) for <code>vslcconvexecx</code> and <code>vslccorreexecx</code> ,	
	COMPLEX(KIND=8), DIMENSION (*) for <code>vslzconvexecx</code> and <code>vslzcorreexecx</code>	
<code>ystride</code> , <code>zstride</code>	INTEGER, DIMENSION (*)	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<code>z</code>	REAL*8 for <code>vsldconvexecx</code> and <code>vsldcorreexecx</code> , COMPLEX*8 for <code>vslcconvexecx</code> and <code>vslccorreexecx</code> , COMPLEX*16 for <code>vslzconvexecx</code> and <code>vslzcorreexecx</code> REAL(KIND=4), DIMENSION(*) for <code>vslsconvexecx</code> and <code>vslscorreexecx</code> , REAL(KIND=8), DIMENSION(*) for <code>vsldconvexecx</code> and <code>vsldcorreexecx</code> , COMPLEX(KIND=4), DIMENSION (*) for <code>vslcconvexecx</code> and <code>vslccorreexecx</code> , COMPLEX(KIND=8), DIMENSION (*) for <code>vslzconvexecx</code> and <code>vslzcorreexecx</code>	Pointer to the array that stores output data. See Data Allocation for more information.
<code>status</code>	INTEGER	Set to <code>VSL_STATUS_OK</code> if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExecX/vslCorrExecX` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array `x`.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTaskX/vslCorrNewTaskX` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

vslConvExecX1D/vslCorrExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

```
status = vslsconvexecx1d(task, y, ystride, z, zstride)
status = vsldconvexecx1d(task, y, ystride, z, zstride)
status = vslcconvexecx1d(task, y, ystride, z, zstride)
status = vslzconvexecx1d(task, y, ystride, z, zstride)
status = vslscorrexecx1d(task, y, ystride, z, zstride)
status = vsldcorrexecx1d(task, y, ystride, z, zstride)
status = vslccorrexecx1d(task, y, ystride, z, zstride)
status = vslzcorrexecx1d(task, y, ystride, z, zstride)
```

Include Files

- mkl.fi, mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <i>task</i> (2) for vslscorrexecx1d, vsldcorrexecx1d, vslccorrexecx1d, vslzcorrexecx1d TYPE (VSL_CONV_TASK) for vslsconvexecx1d, vsldconvexecx1d, vslcconvexecx1d, vslzconvexecx1d TYPE (VSL_CORR_TASK) for vslscorrexecx1d, vsldcorrexecx1d, vslccorrexecx1d, vslzcorrexecx1d VSLCorrTaskPtr for vslsCorrExecX1D, vsldCorrExecX1D, vslcCorrExecX1D, vslzCorrExecX1D	Pointer to the task descriptor.
<i>x, y</i>	REAL*8 for vsldconvexecx1d and vsldcorrexecx1d,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	COMPLEX*8 for <code>vslcconvexecx1d</code> and <code>vslccorrexecx1d</code> ,	
	COMPLEX*16 for <code>vslzconvexecx1d</code> and <code>vslzcorrexecx1d</code>	
	REAL(KIND=4), DIMENSION(*) for <code>vslsconvexecx1d</code> and <code>vslscorrexecx1d</code> ,	
	REAL(KIND=8), DIMENSION(*) for <code>vsldconvexecx1d</code> and <code>vsldcorrexecx1d</code> ,	
	COMPLEX(KIND=4), DIMENSION (*) for <code>vslcconvexecx1d</code> and <code>vslccorrexecx1d</code> ,	
	COMPLEX(KIND=8), DIMENSION (*) for <code>vslzconvexecx1d</code> and <code>vslzcorrexecx1d</code>	
<i>ystride</i> , <i>zstride</i>	INTEGER	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	REAL*8 for <code>vsldconvexecx1d</code> and <code>vsldcorrexecx1d</code> ,	Pointer to the array that stores output data. See Data Allocation for more information.
	COMPLEX*8 for <code>vslcconvexecx1d</code> and <code>vslccorrexecx1d</code> ,	
	COMPLEX*16 for <code>vslzconvexecx1d</code> and <code>vslzcorrexecx1d</code>	
	REAL(KIND=4), DIMENSION(*) for <code>vslsconvexecx1d</code> and <code>vslscorrexecx1d</code> ,	
	REAL(KIND=8), DIMENSION(*) for <code>vsldconvexecx1d</code> and <code>vsldcorrexecx1d</code> ,	
	COMPLEX(KIND=4), DIMENSION (*) for <code>vslcconvexecx1d</code> and <code>vslccorrexecx1d</code> ,	
	COMPLEX(KIND=8), DIMENSION (*) for <code>vslzconvexecx1d</code> and <code>vslzcorrexecx1d</code>	

Name	Type	Description
<i>status</i>	INTEGER	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExecX1D/vslCorrExecX1D` routines computes convolution or correlation of one-dimensional (assuming that `dims = 1`) data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor and pointed to by `task`. If `task` is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

`vslConvDeleteTask/vslCorrDeleteTask`

Destroys the task object and frees the memory.

Syntax

```
errcode = vslconvdeletetask(task)
```

```
errcode = vslcorrdeletetask(task)
```

Include Files

- `mkl.fi, mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>task</i>	INTEGER*4 <code>task(2)</code> for <code>vslcorrdeletetask</code> TYPE (VSL_CONV_TASK) for <code>vslconvdeletetask</code> TYPE (VSL_CORR_TASK) for <code>vslcorrdeletetask</code>	Pointer to the task descriptor.

Output Parameters

Name	Type	Description
<i>errcode</i>	INTEGER	Contains 0 if the task object is deleted successfully. Contains an error code if an error occurred.

Description

The `vslConvDeleteTask/vslCorrvDeleteTask` routine deletes the task descriptor object and frees any working memory and the memory allocated for the data structure. The task pointer is set to `NULL`.

Note that if the `vslConvDeleteTask/vslCorrvDeleteTask` routine does not delete the task successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

NOTE

You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine terminates with no system crash.

Task Copy

The routines are designed for copying convolution and correlation task descriptors.

`vslConvCopyTask/vslCorrCopyTask`

Copies a descriptor for convolution or correlation task.

Syntax

```
status = vslconvcopytask(newtask, srctask)
```

```
status = vslcorrcopytask(newtask, srctask)
```

Include Files

- `mkl.fi`, `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>srctask</code>	INTEGER*4 <code>srctask(2)</code> for <code>vslcorrcopytask</code> TYPE (VSL_CONV_TASK) for <code>vslconvcopytask</code> TYPE (VSL_CORR_TASK) for <code>vslcorrcopytask</code>	Pointer to the source task descriptor.

Output Parameters

Name	Type	Description
<code>newtask</code>	INTEGER*4 <code>srctask(2)</code> for <code>vslcorrcopytask</code> TYPE (VSL_CONV_TASK) for <code>vslconvcopytask</code> TYPE (VSL_CORR_TASK) for <code>vslcorrcopytask</code>	Pointer to the new task descriptor.
<code>status</code>	INTEGER	Current status of the source task.

Description

If a task object *srctask* already exists, you can use an appropriate `vslConvCopyTask/vslCorrCopyTask` routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see [Introduction to Convolution and Correlation](#) for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Usage Examples

This section demonstrates how you can use the Intel MKL routines to perform some common convolution and correlation operations both for single-threaded and multithreaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single-threaded calculations and can be used with C or C++ compilers.

Function `scond1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int scond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task, VSL_CONV_MODE_DIRECT, nh, nx, ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h, inch, x, incx, y, incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Function `sconf1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int sconf1(
    int init,
    float h[], int inclh,
    float x[], int inclx, int inc2x,
    float y[], int incly, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;

    /* assume that aux1!=0 and naux1 is big enough */
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;

    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task, VSL_CONV_MODE_FFT,
            nh, nx, ny, h, inclh);

    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);

        for (i=0; i<m; i++) {
            float* xi = &x[inc2x * i];
            float* yi = &y[inc2y * i];

            /* task is implicitly committed at i==0 */
            status = vslsConvExecX1D(*task, xi, inclx, yi, incly);
        };
    };

    vslConvDeleteTask(task);

    return status;
}
```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If $m > 1$, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create m copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using [Task Editors](#)) before copying it.

The sample code in this case may look as follows:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    . . .
    vslConvSetStart(*task, &iy0);
    . . .
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    . . .
```

Then, *m* threads may be started to execute different copies of the task:

```
. . .
    float* xi = &x[inc2x * i];
    float* yi = &y[inc2y * i];
    ss[i]=vslsConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
    . . .
```

And finally, after all threads have finished the calculations, overall status should be collected from all task objects. The following code signals the first error found, if any:

```
. . .
    for (i=0; i<m; i++) {
        status = ss[i];
        if (status != 0) /* 0 means "OK" */
            break;
    };
    return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. That is why different threads must use different copies of the task.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Optimization Notice
Notice revision #20110804

Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbf{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbf{Z} = \{0, \pm 1, \pm 2, \dots\}$	The set of integer numbers.
$\mathbf{Z}^N = \mathbf{Z} \times \dots \times \mathbf{Z}$	The set of N-dimensional series of integer numbers.
$p = (p_1, \dots, p_N) \in \mathbf{Z}^N$	N-dimensional series of integers.
$u: \mathbf{Z}^N \rightarrow \mathbf{R}$	Function u with arguments from \mathbf{Z}^N and values from \mathbf{R} .
$u(p) = u(p_1, \dots, p_N)$	The value of the function u for the argument (p_1, \dots, p_N) .
$w = u * v$	Function w is the convolution of the functions u, v .
$w = u \bullet v$	Function w is the correlation of the functions u, v .

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1, \dots, N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1, \dots, N$
- series $r = \sup\{p, q\}$ is defines as $r^n = \max\{p^n, q^n\}$ for every $n=1, \dots, N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1, \dots, N$
- inequality $p \leq q$ means that $p^n \leq q^n$ for every $n=1, \dots, N$.

A function $u(p)$ is called a finite function if there exist series $p^{\min}, p^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0$$

implies

$$p^{\min} \leq p \leq p^{\max}.$$

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series $p^{\min}, p^{\max}, q^{\min}, q^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0 \text{ implies } p^{\min} \leq p \leq p^{\max}.$$

$$v(q) \neq 0 \text{ implies } q^{\min} \leq q \leq q^{\max}.$$

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u * v$ is the convolution of u and v , then:

$$w(r) \neq 0 \text{ implies } \mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max},$$

$$\text{where } \mathbf{R}^{\min} = p^{\min} + Q^{\min} \text{ and } \mathbf{R}^{\max} = p^{\max} + Q^{\max}.$$

If $\mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max}$, then:

$$w(r) = \sum u(t) \cdot v(r-t) \text{ is the sum for all } t \in \mathbf{Z}^N \text{ such that } \mathbf{T}^{\min} \leq t \leq \mathbf{T}^{\max},$$

$$\text{where } \mathbf{T}^{\min} = \sup\{p^{\min}, r - Q^{\max}\} \text{ and } \mathbf{T}^{\max} = \inf\{p^{\max}, r - Q^{\min}\}.$$

Linear Correlation

If function $w = u \bullet v$ is the correlation of u and v , then:

$$w(r) \neq 0 \text{ implies } \mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max},$$

$$\text{where } \mathbf{R}^{\min} = Q^{\min} - p^{\max} \text{ and } \mathbf{R}^{\max} = Q^{\max} - p^{\min}.$$

If $\mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max}$, then:

$w(r) = \sum u(t) \cdot v(r+t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{\min} \leq t \leq \mathbf{T}^{\max}$,
 where $\mathbf{T}^{\min} = \sup\{P^{\min}, Q^{\min}-r\}$ and $\mathbf{T}^{\max} = \inf\{P^{\max}, Q^{\max}-r\}$.

Representation of the functions u , v , w as the input/output data for the Intel MKL convolution and correlation functions is described in the [Data Allocation](#) section below.

Data Allocation

This section explains the relation between:

- mathematical finite functions u , v , w introduced in the section [Mathematical Notation and Definitions](#);
- multi-dimensional input and output data vectors representing the functions u , v , w ;
- arrays u , v , w used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays x , y , z
- Shape arrays $xshape$, $yshape$, $zshape$
- Strides within arrays $xstride$, $ystride$, $zstride$
- Parameters $start$, $decimation$

Finite Functions and Data Vectors

The finite functions $u(p)$, $v(q)$, and $w(r)$ introduced above are represented as multi-dimensional vectors of input and output data:

input u (i_1, \dots, i_{dims}) for $u(p_1, \dots, p_N)$

input v (j_1, \dots, j_{dims}) for $v(q_1, \dots, q_N)$

output(k_1, \dots, k_{dims}) for $w(r_1, \dots, r_N)$.

Parameter $dims$ represents the number of dimensions and is equal to N .

The parameters $xshape$, $yshape$, and $zshape$ define the shapes of input/output vectors:

input u (i_1, \dots, i_{dims}) is defined if $1 \leq i_n \leq xshape(n)$ for every $n=1, \dots, dims$

input v (j_1, \dots, j_{dims}) is defined if $1 \leq j_n \leq yshape(n)$ for every $n=1, \dots, dims$

output(k_1, \dots, k_{dims}) is defined if $1 \leq k_n \leq zshape(n)$ for every $n=1, \dots, dims$.

Relation between the input vectors and the functions u and v is defined by the following formulas:

input u (i_1, \dots, i_{dims}) = $u(p_1, \dots, p_N)$, where $p_n = P_n^{\min} + (i_n - 1)$ for every n

input v (j_1, \dots, j_{dims}) = $v(q_1, \dots, q_N)$, where $q_n = Q_n^{\min} + (j_n - 1)$ for every n .

The relation between the output vector and the function $w(r)$ is similar (but only in the case when parameters $start$ and $decimation$ are not defined):

output(k_1, \dots, k_{dims}) = $w(r_1, \dots, r_N)$, where $r_n = R_n^{\min} + (k_n - 1)$ for every n .

If the parameter $start$ is defined, it must belong to the interval $R_n^{\min} \leq start(n) \leq R_n^{\max}$. If defined, the $start$ parameter replaces R_n^{\min} in the formula:

output(k_1, \dots, k_{dims}) = $w(r_1, \dots, r_N)$, where $r_n = start(n) + (k_n - 1)$

If the parameter $decimation$ is defined, it changes the relation according to the following formula:

output(k_1, \dots, k_{dims}) = $w(r_1, \dots, r_N)$, where $r_n = R_n^{\min} + (k_n - 1) * decimation(n)$

If both parameters $start$ and $decimation$ are defined, the formula is as follows:

output(k_1, \dots, k_{dims}) = $w(r_1, \dots, r_N)$, where $r_n = start(n) + (k_n - 1) * decimation(n)$

The convolution and correlation software checks the values of *zshape*, *start*, and *decimation* during task commitment. If r_n exceeds R_n^{\max} for some $k_n, n=1, \dots, \text{dims}$, an error is raised.

Allocation of Data Vectors

Both parameter arrays *x* and *y* contain input data vectors in memory, while array *z* is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters *x*, *y*, and *z*, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays *x*, *y*, and *z* is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

`inputu(i1, ..., idims)` is allocated at `x(e)`
`inputv(j1, ..., jdims)` is allocated at `y(f)`
`output(k1, ..., kdims)` is allocated at `z(g)`.

The indices *e*, *f*, and *g* are defined as follows:

$e = 1 + \sum x\text{stride}(n) \cdot dx(n)$ (the sum is for all $n=1, \dots, \text{dims}$)
 $f = 1 + \sum y\text{stride}(n) \cdot dy(n)$ (the sum is for all $n=1, \dots, \text{dims}$)
 $g = 1 + \sum z\text{stride}(n) \cdot dz(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

The distances $dx(n)$, $dy(n)$, and $dz(n)$ depend on the signum of the stride:

$dx(n) = i_n - 1$ if $x\text{stride}(n) > 0$, or $dx(n) = i_n - x\text{shape}(n)$ if $x\text{stride}(n) < 0$
 $dy(n) = j_n - 1$ if $y\text{stride}(n) > 0$, or $dy(n) = j_n - y\text{shape}(n)$ if $y\text{stride}(n) < 0$
 $dz(n) = k_n - 1$ if $z\text{stride}(n) > 0$, or $dz(n) = k_n - z\text{shape}(n)$ if $z\text{stride}(n) < 0$

The definitions of indices *e*, *f*, and *g* assume that indexes for arrays *x*, *y*, and *z* are started from unity:

x(*e*) is defined for $e=1, \dots, \text{length}(x)$
y(*f*) is defined for $f=1, \dots, \text{length}(y)$
z(*g*) is defined for $g=1, \dots, \text{length}(z)$

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array *z* for one-dimensional and two-dimensional cases.

One-dimensional case. If $\text{dims}=1$, then *zshape* is the number of the output values to be stored in the array *z*. The actual length of array *z* may be greater than *zshape* elements.

If $z\text{stride} > 1$, output values are stored with the stride: `output(1)` is stored to `z(1)`, `output(2)` is stored to `z(1+zstride)`, and so on. Hence, the actual length of *z* must be at least $1+z\text{stride} \cdot (z\text{shape}-1)$ elements or more.

If $z\text{stride} < 0$, it still defines the stride between elements of array *z*. However, the order of the used elements is the opposite. For the *k*-th output value, `output(k)` is stored in `z(1+|zstride|*(zshape-k))`, where $|z\text{stride}|$ is the absolute value of *zstride*. The actual length of the array *z* must be at least $1+|z\text{stride}| \cdot (z\text{shape} - 1)$ elements.

Two-dimensional case. If $\text{dims}=2$, the output data is a two-dimensional matrix. The value `zstride(1)` defines the stride inside matrix columns, that is, the stride between the `output(k1, k2)` and `output(k1+1, k2)` for every pair of indices k_1, k_2 . On the other hand, `zstride(2)` defines the stride between columns, that is, the stride between `output(k1, k2)` and `output(k1, k2+1)`.

If `zstride(2)` is greater than `zshape(1)`, this causes sparse allocation of columns. If the value of `zstride(2)` is smaller than `zshape(1)`, this may result in the transposition of the output matrix. For example, if `zshape = (2,3)`, you can define `zstride = (3,1)` to allocate output values like transposed matrix of the shape `3x2`.

Whether `zstride` assumes this kind of transformations or not, you need to ensure that different elements output ($k_1, \dots, k_{\text{dims}}$) will be stored in different locations $z(g)$.

Summary Statistics

The Summary Statistics domain comprises a set of routines that compute basic statistical estimates for single and double precision multi-dimensional datasets.

See the definition of the supported operations in the [Mathematical Notation and Definitions](#) section.

The Summary Statistics routines calculate:

- raw and central moments up to the fourth order
- skewness and excess kurtosis (further referred to as *kurtosis* for brevity)
- variation coefficient
- quantiles and order statistics
- minimum and maximum
- variance-covariance/correlation matrix
- pooled/group variance-covariance matrix and mean
- partial variance-covariance/correlation matrix
- robust estimators for variance-covariance matrix and mean in presence of outliers
- raw/central partial sums up to the fourth order (for brevity referred to as *raw/central sums*)
- matrix of cross-products and sums of squares (for brevity referred to as *cross-product matrix*)
- median absolute deviation, mean absolute deviation

The library also contains functions to perform the following tasks:

- Detect outliers in datasets
- Support missing values in datasets
- Parameterize correlation matrices
- Compute quantiles for streaming data

You can access the Summary Statistics routines through the Fortran 90 and C89 language interfaces. You can use the Fortran 90 interface with programs written in Fortran 95.

Intel MKL provides the `mk1_vs1.f90` header file. All header files are in the directory

```
{MKL}/include
```

See more details about the Fortran header in the [Random Number Generators](#) section of this chapter.

You can find examples that demonstrate calculation of the Summary Statistics estimates in the examples directories:

```
{MKL}/examples/vs1f
```

The Summary Statistics API is implemented through task objects, or tasks. A task object is a data structure, or a descriptor, holding parameters that determine a specific Summary Statistics operation. For example, such parameters may be precision, dimensions of user data, the matrix of the observations, or shapes of data arrays.

All the Summary Statistics routines process a task object as follows:

1. Create a task.
2. Modify settings of the task parameters.
3. Compute statistical estimates.
4. Destroy the task.

The Summary Statistics functions fall into the following categories:

Task Constructors - routines that create a new task object descriptor and set up most common parameters (dimension, number of observations, and matrix of the observations).

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Computation Routine - a routine that computes specified statistical estimates.

Task Destructor - a routine that deletes the task object and frees the memory.

A Summary Statistics task object contains a series of pointers to the input and output data arrays. You can read and modify the datasets and estimates at any time but you should allocate and release memory for such data.

See detailed information on the algorithms, API, and their usage in the *Intel® MKL Summary Statistics Application Notes* [SS Notes].

Naming Conventions

The names of routines in the Summary Statistics are in lowercase (*vslsssedigitquantiles*), while the names of types and constants are in uppercase. The names are not case-sensitive.

The names of routines have the following structure:

```
vsl[datatype]ss<base name>
```

where

- *vsl* is a prefix indicating that the routine belongs to Intel MKL Vector Statistics.
- *[datatype]* specifies the type of the input and/or output data and can be *s* (single precision real type), *d* (double precision real type), or *i* (integer type).
- *ss/ss* indicates that the routine is intended for calculations of the Summary Statistics estimates.
- *<base name>* specifies a particular functionality that the routine is designed for, for example, *NewTask*, *Compute*, *DeleteTask*.

NOTE

The Summary Statistics routine *vslDeleteTask* for deletion of the task is independent of the data type and its name omits the *[datatype]* field.

Data Types

The Summary Statistics routines use the following data types for the calculations:

Type	Data Object
Fortran 90: TYPE (VSL_SS_TASK)	Pointer to a Summary Statistics task
Fortran 90: REAL (KIND=4)	Input/output user data in single precision
Fortran 90: REAL (KIND=8)	Input/output user data in double precision
Fortran 90: INTEGER or INTEGER (KIND=8)	Other data

NOTE

The actual size of the generic integer type is platform-specific and can be 32 or 64 bits in length. Before you compile your application, set an appropriate size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® MKL User's Guide*.

Parameters

The basic parameters in the task descriptor (addresses of dimensions, number of observations, and datasets) are assigned values when the task editors create or modify the task object. Other parameters are determined by the specific task and changed by the task editors.

Task Status and Error Reporting

The task status is an integer value, which is zero if no error is detected, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The header files define symbolic names for the status codes. These names are defined as integer constants via the `PARAMETER` operators.

If no error is detected, the function returns the `VSL_STATUS_OK` code, which is defined as zero:

```
INTEGER, PARAMETER::VSL_STATUS_OK = 0
```

In the case of an error, the function returns a non-zero error code that specifies the origin of the failure. The header files define the following status codes for the Summary Statistics error codes:

Summary Statistics Status Codes

Status Code	Description
<code>VSL_STATUS_OK</code>	Operation is successfully completed.
<code>VSL_SS_ERROR_ALLOCATION_FAILURE</code>	Memory allocation has failed.
<code>VSL_SS_ERROR_BAD_DIMEN</code>	Dimension value is invalid.
<code>VSL_SS_ERROR_BAD_OBSERV_N</code>	Invalid number (zero or negative) of observations was obtained.
<code>VSL_SS_ERROR_STORAGE_NOT_SUPPORTED</code>	Storage format is not supported.
<code>VSL_SS_ERROR_BAD_INDC_ADDR</code>	Array of indices is not defined.
<code>VSL_SS_ERROR_BAD_WEIGHTS</code>	Array of weights contains negative values.
<code>VSL_SS_ERROR_BAD_MEAN_ADDR</code>	Array of means is not defined.
<code>VSL_SS_ERROR_BAD_2R_MOM_ADDR</code>	Array of the second order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_3R_MOM_ADDR</code>	Array of the third order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_4R_MOM_ADDR</code>	Array of the fourth order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_2C_MOM_ADDR</code>	Array of the second order central moments is not defined.
<code>VSL_SS_ERROR_BAD_3C_MOM_ADDR</code>	Array of the third order central moments is not defined.
<code>VSL_SS_ERROR_BAD_4C_MOM_ADDR</code>	Array of the fourth order central moments is not defined.
<code>VSL_SS_ERROR_BAD_KURTOSIS_ADDR</code>	Array of kurtosis values is not defined.
<code>VSL_SS_ERROR_BAD_SKEWNESS_ADDR</code>	Array of skewness values is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_MIN_ADDR	Array of minimum values is not defined.
VSL_SS_ERROR_BAD_MAX_ADDR	Array of maximum values is not defined.
VSL_SS_ERROR_BAD_VARIATION_ADDR	Array of variation coefficients is not defined.
VSL_SS_ERROR_BAD_COV_ADDR	Covariance matrix is not defined.
VSL_SS_ERROR_BAD_COR_ADDR	Correlation matrix is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER_ADDR	Array of quantile orders is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER	Quantile order value is invalid.
VSL_SS_ERROR_BAD_QUANT_ADDR	Array of quantiles is not defined.
VSL_SS_ERROR_BAD_ORDER_STATS_ADDR	Array of order statistics is not defined.
VSL_SS_ERROR_MOMORDER_NOT_SUPPORTED	Moment of requested order is not supported.
VSL_SS_NOT_FULL_RANK_MATRIX	Correlation matrix is not of full rank.
VSL_SS_ERROR_ALL_OBSERVS_OUTLIERS	All observations are outliers. (At least one observation must not be an outlier.)
VSL_SS_ERROR_BAD_ROBUST_COV_ADDR	Robust covariance matrix is not defined.
VSL_SS_ERROR_BAD_ROBUST_MEAN_ADDR	Array of robust means is not defined.
VSL_SS_ERROR_METHOD_NOT_SUPPORTED	Requested method is not supported.
VSL_SS_ERROR_NULL_TASK_DESCRIPTOR	Task descriptor is null.
VSL_SS_ERROR_BAD_OBSERV_ADDR	Dataset matrix is not defined.
VSL_SS_ERROR_BAD_ACCUM_WEIGHT_ADDR	Pointer to the variable that holds the value of accumulated weight is not defined.
VSL_SS_ERROR_SINGULAR_COV	Covariance matrix is singular.
VSL_SS_ERROR_BAD_POOLED_COV_ADDR	Pooled covariance matrix is not defined.
VSL_SS_ERROR_BAD_POOLED_MEAN_ADDR	Array of pooled means is not defined.
VSL_SS_ERROR_BAD_GROUP_COV_ADDR	Group covariance matrix is not defined.
VSL_SS_ERROR_BAD_GROUP_MEAN_ADDR	Array of group means is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC_ADDR	Array of group indices is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC	Group indices have improper values.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_ADDR	Array of parameters for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_N_ADDR	Pointer to size of the parameter array for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_WEIGHTS_ADDR	Output of the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_ADDR	Array of parameters of the robust covariance estimation algorithm is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_N_ADDR	Pointer to the number of parameters of the algorithm for robust covariance is not defined.
VSL_SS_ERROR_BAD_STORAGE_ADDR	Pointer to the variable that holds the storage format is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX_ADDR	Array that encodes sub-components of a random vector for the partial covariance algorithm is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX	Array that encodes sub-components of a random vector for partial covariance has improper values.
VSL_SS_ERROR_BAD_PARTIAL_COV_ADDR	Partial covariance matrix is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COR_ADDR	Partial correlation matrix is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_ADDR	Array of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_N_ADDR	Pointer to number of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_BAD_PARAMS_N	Size of the parameter array of the Multiple Imputation method is invalid.
VSL_SS_ERROR_BAD_MI_PARAMS	Parameters of the Multiple Imputation method are invalid.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_N_ADDR	Pointer to the number of initial estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_ADDR	Array of initial estimates for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_ADDR	Array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N_ADDR	Pointer to the size of the array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N_ADDR	Pointer to the number of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_ADDR	Array of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N	Invalid size of the array of simulated values in the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N	Invalid size of an array to hold parameter estimates obtained using the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_OUTPUT_PARAMS	Array of output parameters in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_N_ADDR	Pointer to the number of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_ADDR	Array of prior parameters is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_MI_MISSING_VALS_N	Invalid number of missing values was obtained.
VSL_SS_SEMIDEFINITE_COR	Correlation matrix passed into the parameterization function is semi-definite.
VSL_SS_ERROR_BAD_PARAMTR_COR_ADDR	Correlation matrix to be parameterized is not defined.
VSL_SS_ERROR_BAD_COR	All eigenvalues of the correlation matrix to be parameterized are non-positive.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N_ADDR	Pointer to the number of parameters for the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_ADDR	Array of parameters of the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N	Invalid number of parameters of the quantile computation algorithm for streaming data has been obtained.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS	Invalid parameters of the quantile computation algorithm for streaming data have been passed.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER_ADDR	Array of the quantile orders for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER	Invalid quantile order for streaming data is defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ADDR	Array of quantiles for streaming data is not defined.
VSL_SS_ERROR_BAD_SUM_ADDR	Array of sums is not defined.
VSL_SS_ERROR_BAD_2R_SUM_ADDR	Array of raw sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3R_SUM_ADDR	Array of raw sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4R_SUM_ADDR	Array of raw sums of 4th order is not defined.
VSL_SS_ERROR_BAD_2C_SUM_ADDR	Array of central sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3C_SUM_ADDR	Array of central sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4C_SUM_ADDR	Array of central sums of 4th order is not defined.
VSL_SS_ERROR_BAD_CP_SUM_ADDR	Cross-product matrix is not defined.
VSL_SS_ERROR_BAD_MDAD_ADDR	Array of median absolute deviations is not defined.
VSL_SS_ERROR_BAD_MNAD_ADDR	Array of mean absolute deviations is not defined.
VSL_SS_ERROR_BAD_SORTED_OBSERV_ADDR	Array for storing observation sorting results is not defined.

Status Code	Description
VSL_SS_ERROR_ERROR_INDICES_NOT_SUPPORTED	Array of indices is not supported.

Routines for robust covariance estimation, outlier detection, partial covariance estimation, multiple imputation, and parameterization of a correlation matrix can return internal error codes that are related to a specific implementation. Such error codes indicate invalid input data or other bugs in the Intel MKL routines other than the Summary Statistics routines.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters.

NOTE

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

vslSSNewTask

Creates and initializes a new summary statistics task descriptor.

Syntax

```
status = vslsssnewtask(task, p, n, xstorage, x, w, indices)
```

```
status = vslldssnewtask(task, p, n, xstorage, x, w, indices)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>p</i>	Fortran: INTEGER	Dimension of the task, number of variables
<i>n</i>	Fortran: INTEGER	Number of observations
<i>xstorage</i>	Fortran: INTEGER	Storage format of matrix of observations
<i>x</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <code>vslsssnewtask</code> REAL(KIND=8) DIMENSION(*) for <code>vslldssnewtask</code>	Matrix of observations
<i>w</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <code>vslsssnewtask</code> REAL(KIND=8) DIMENSION(*) for <code>vslldssnewtask</code>	Array of weights of size <i>n</i> . Elements of the arrays are non-negative numbers. If a <code>NULL</code> pointer is passed, each observation is assigned weight equal to 1.

Name	Type	Description
<i>indices</i>	Fortran: INTEGER, DIMENSION(*)	Array of vector components that will be processed. Size of array is p . If a NULL pointer is passed, all components of random vector are processed.

Output Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK)	Descriptor of the task
<i>status</i>	Fortran: INTEGER	Set to VSL_STATUS_OK if the task is created successfully, otherwise a non-zero error code is returned.

Description

Each `vslSSNewTask` constructor routine creates a new summary statistics task descriptor with the user-specified value for a required parameter, dimension of the task. The optional parameters (matrix of observations, its storage format, number of observations, weights of observations, and indices of the random vector components) are set to their default values.

The observations of random p -dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, which are n vectors of dimension p , are passed as a one-dimensional array x . The parameter `xstorage` defines the storage format of the observations and takes one of the possible values listed in [Table "Storage format of matrix of observations and order statistics"](#).

Storage format of matrix of observations, order statistics, and matrix of sorted observations

Parameter	Description
VSL_SS_MATRIX_STORAGE_ROWS	The observations of random vector ξ are packed by rows: n data points for the vector component ξ_1 come first, n data points for the vector component ξ_2 come second, and so forth.
VSL_SS_MATRIX_STORAGE_COLS	The observations of random vector ξ are packed by columns: the first p -dimensional observation of the vector ξ comes first, the second p -dimensional observation of the vector comes second, and so forth.

NOTE

Since matrices in Fortran are stored by columns while in C they are stored by rows, initialization of the `xstorage` variable in Fortran is opposite to that in C. Set `xstorage` to `VSL_SS_MATRIX_STORAGE_COLS`, if the dataset is stored as a two-dimensional matrix that consists of p rows and n columns; otherwise, use the `VSL_SS_MATRIX_STORAGE_ROWS` constant.

A one-dimensional array w of size n contains non-negative weights assigned to the observations. You can pass a NULL array into the constructor. In this case, each observation is assigned the default value of the weight.

You can choose vector components for which you wish to compute statistical estimates. If an element of the vector `indices` of size p contains 0, the observations that correspond to this component are excluded from the calculations. If you pass the NULL value of the parameter into the constructor, statistical estimates for all random variables are computed.

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

Task Editors

Task editors are intended to set up or change the task parameters listed in [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#). As an example, to compute the sample mean for a one-dimensional dataset, initialize a variable for the mean value, and pass its address into the task as shown in the example below:

```
#define DIM 1
#define N 1000

int main()
{
    VSLSSTaskPtr task;
    double x[N];
    double mean;
    MKL_INT p, n, xstorage;
    int status;
    /* initialize variables used in the computations of sample mean */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
    mean = 0.0;

    /* create task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* initialize task parameters */
    status = vsldSSEditTask( task, VSL_SS_ED_MEAN, &mean );

    /* compute mean using SS fast method */
    status = vsldSSCompute(task, VSL_SS_MEAN, VSL_SS_METHOD_FAST );

    /* deallocate task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

Use the single (`vsldSSeditTask`) or double (`vsldSSeditTask`) version of an editor, to initialize single or double precision version task parameters, respectively. Use an integer version of an editor (`vsldSSeditTask`) to initialize parameters of the integer type.

[Table "Summary Statistics Task Editors"](#) lists the task editors for Summary Statistics. Each of them initializes and/or modifies a respective group of related parameters.

Summary Statistics Task Editors

Editor	Description
<code>vsldSSeditTask</code>	Changes a pointer in the task descriptor.
<code>vsldSSeditMoments</code>	Changes pointers to arrays associated with raw and central moments.
<code>vsldSSeditSums</code>	Modifies the pointers to arrays that hold sum estimates.
<code>vsldSSeditCovCor</code>	Changes pointers to arrays associated with covariance and/or correlation matrices.
<code>vsldSSeditCP</code>	Modifies the pointers to cross-product matrix parameters.

Editor	Description
<code>vslSSEditPartialCovCor</code>	Changes pointers to arrays associated with partial covariance and/or correlation matrices.
<code>vslSSEditQuantiles</code>	Changes pointers to arrays associated with quantile/order statistics calculations.
<code>vslSSEditStreamQuantiles</code>	Changes pointers to arrays for quantile related calculations for streaming data.
<code>vslSSEditPooledCovariance</code>	Changes pointers to arrays associated with algorithms related to a pooled covariance matrix.
<code>vslSSEditRobustCovariance</code>	Changes pointers to arrays for robust estimation of a covariance matrix and mean.
<code>vslSSEditOutliersDetection</code>	Changes pointers to arrays for detection of outliers.
<code>vslSSEditMissingValues</code>	Changes pointers to arrays associated with the method of supporting missing values in a dataset.
<code>vslSSEditCorParameterization</code>	Changes pointers to arrays associated with the algorithm for parameterization of a correlation matrix.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslSSEditTask

Modifies address of an input/output parameter in the task descriptor.

Syntax

```
status = vslsssedittask(task, parameter, par_addr)
status = vsldssedittask(task, parameter, par_addr)
status = vslissedittask(task, parameter, par_addr)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE (VSL_SS_TASK)</code>	Descriptor of the task
<code>parameter</code>	Fortran: <code>INTEGER</code>	Parameter to change
<code>par_addr</code>	Fortran: <code>REAL (KIND=4) DIMENSION (*)</code> for <code>vslsssedittask</code> <code>REAL (KIND=8) DIMENSION (*)</code> for <code>vsldssedittask</code>	Address of the new parameter

Name	Type	Description
	INTEGER DIMENSION(*) for vslissedittask	

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSeditTask` routine replaces the pointer to the parameter stored in the Summary Statistics task descriptor with the `par_addr` pointer. If you pass the `NULL` pointer to the editor, no changes take place in the task and a corresponding error code is returned. See [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#) for the predefined values of the parameter.

Use the single (`vslssedittask`) or double (`vslDssedittask`) version of the editor, to initialize single or double precision version task parameters, respectively. Use an integer version of the editor (`vslissedittask`) to initialize parameters of the integer type.

Parameters of Summary Statistics Task to Be Initialized or Modified

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_DIMEN	i	Address of a variable that holds the task dimension	Required. Positive integer value.
VSL_SS_ED_OBSERV_N	i	Address of a variable that holds the number of observations	Required. Positive integer value.
VSL_SS_ED_OBSERV	d, s	Address of the observation matrix	Required. Provide the matrix containing your observations.
VSL_SS_ED_OBSERV_STORAGE	i	Address of a variable that holds the storage format for the observation matrix	Required. Provide a storage format supported by the library whenever you pass a matrix of observations. ¹
VSL_SS_ED_INDC	i	Address of the array of indices	Optional. Provide this array if you need to process individual components of the random vector. Set entry <i>i</i> of the array to one to include the <i>i</i> th coordinate in the analysis. Set entry <i>i</i> of the array to zero to exclude the <i>i</i> th coordinate from the analysis.
VSL_SS_ED_WEIGHTS	d, s	Address of the array of observation weights	Optional. If the observations have weights different from the default weight (one), set entries of the array to non-negative floating point values.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_MEAN	d, s	Address of the array of means	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_MOM	d, s	Address of an array of raw moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_MOM	d, s	Address of an array of raw moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_MOM	d, s	Address of an array of raw moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_MOM	d, s	Address of an array of central moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_3C_MOM	d, s	Address of an array of central moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_4C_MOM	d, s	Address of an array of central moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_KURTOSIS	d, s	Address of the array of kurtosis estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise,

Parameter Value	Type	Purpose	Initialization
			do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_SKEWNESS	d, s	Address of the array of skewness estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_MIN	d, s	Address of the array of minimum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_MAX	d, s	Address of the array of maximum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_VARIATION	d, s	Address of the array of variation coefficients	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_COV	d, s	Address of a covariance matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Make sure you also provide an array for the mean.
VSL_SS_ED_COV_STORAGE	i	Address of the variable that holds the storage format for a covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute the covariance matrix. ²
VSL_SS_ED_COR	d, s	Address of a correlation matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. If you initialize the matrix in non-trivial way, make sure that the main diagonal contains variance values. Also, provide an array for the mean.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_COR_STORAGE	i	Address of the variable that holds the correlation storage format for a correlation matrix	Required. Provide a storage format supported by the library whenever you intend to compute the correlation matrix. ²
VSL_SS_ED_ACCUM_WEIGHT	d, s	Address of the array of size 2 that holds the accumulated weight (sum of weights) in the first position and the sum of weights squared in the second position	Optional. Set the entries of the matrix to meaningful values (typically zero) if you intend to do progressive processing of the dataset or need the sum of weights and sum of squared weights assigned to observations.
VSL_SS_ED_QUANT_ORDER_N	i	Address of the variable that holds the number of quantile orders	Required. Positive integer value. Provide the number of quantile orders whenever you compute quantiles.
VSL_SS_ED_QUANT_ORDER	d, s	Address of the array of quantile orders	Required. Set entries of array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_QUANT_QUANTILE S	d, s	Address of the array of quantiles	None.
VSL_SS_ED_ORDER_STATS	d, s	Address of the array of order statistics	None.
VSL_SS_ED_GROUP_INDC	i	Address of the array of group indices used in computation of a pooled covariance matrix	Required. Set entry i to integer value k if the observation belongs to group k . Values of k take values in the range $[0, g-1]$, where g is the number of groups.
VSL_SS_ED_POOLED_COV_STO RAGE	i	Address of a variable that holds the storage format for a pooled covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute pooled covariance. ²
VSL_SS_ED_POOLED_MEAN	d, s	Address of an array of pooled means	None.
VSL_SS_ED_POOLED_COV	d, s	Address of pooled covariance matrices	None.
VSL_SS_ED_GROUP_COV_INDC	i	Address of an array of indices for which covariance/means should be computed	Optional. Set the k th entry of the array to 1 if you need group covariance and mean for group k ; otherwise set it to zero.
VSL_SS_ED_REQ_GROUP_INDC	i	Address of an array of indices for which group estimates such as covariance or means are requested	Optional. Set the k th entry of the array to 1 if you need an estimate for group k ; otherwise set it to zero.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_GROUP_MEANS	i	Address of an array of group means	None.
VSL_SS_ED_GROUP_COV_STORAGE	d, s	Address of a variable that holds the storage format for a group covariance matrix	Required. Provide a storage format supported by the library whenever you intend to get group covariance. ²
VSL_SS_ED_GROUP_COV	d, s	Address of group covariance matrices	None.
VSL_SS_ED_ROBUST_COV_STORAGE	d, s	Address of a variable that holds the storage format for a robust covariance matrix	Required. Provide a storage format supported by the library whenever you compute robust covariance ² .
VSL_SS_ED_ROBUST_COV_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters of the method for robust covariance estimation	Required. Set to the number of TBS parameters, <i>VSL_SS_TBS_PARAMS_N</i> .
VSL_SS_ED_ROBUST_COV_PARAMS	d, s	Address of an array of parameters of the method for robust estimation of a covariance	Required. Set the entries of the array according to the description in vslSSEditRobustCovariance .
VSL_SS_ED_ROBUST_MEAN	i	Address of an array of robust means	None.
VSL_SS_ED_ROBUST_COV	d, s	Address of a robust covariance matrix	None.
VSL_SS_ED_OUTLIERS_PARAMS_N	d, s	Address of a variable that holds the number of parameters of the outlier detection method	Required. Set to the number of outlier detection parameters, <i>VSL_SS_BACON_PARAMS_N</i> .
VSL_SS_ED_OUTLIERS_PARAMS	i	Address of an array of algorithmic parameters for the outlier detection method	Required. Set the entries of the array according to the description in vslSSEditOutliersDetection .
VSL_SS_ED_OUTLIERS_WEIGHT	d, s	Address of an array of weights assigned to observations by the outlier detection method	None.
VSL_SS_ED_ORDER_STATS_STORAGE	d, s	Address of a variable that holds the storage format of an order statistics matrix	Required. Provide a storage format supported by the library whenever you compute a matrix of order statistics. ¹
VSL_SS_ED_PARTIAL_COV_IDS	i	Address of an array that encodes subcomponents of a random vector	Required. Set the entries of the array according to the description in vslSSEditPartialCovCor .

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_PARTIAL_COV	d, s	Address of a partial covariance matrix	None.
VSL_SS_ED_PARTIAL_COV_STORAGE	i	Address of a variable that holds the storage format of a partial covariance matrix	Required. Provide a storage format supported by the library whenever you compute the partial covariance. ²
VSL_SS_ED_PARTIAL_COR	d, s	Address of a partial correlation matrix	None.
VSL_SS_ED_PARTIAL_COR_STORAGE	i	Address of a variable that holds the storage format for a partial correlation matrix	Required. Provide a storage format supported by the library whenever you compute the partial correlation. ²
VSL_SS_ED_MI_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters for the Multiple Imputation method	Required. Set to the number of MI parameters, <code>VSL_SS_MI_PARAMS_SIZE</code> .
VSL_SS_ED_MI_PARAMS	d, s	Address of an array of algorithmic parameters for the Multiple Imputation method	Required. Set entries of the array according to the description in vslSSEditMissingValues .
VSL_SS_ED_MI_INIT_ESTIMATES_N	i	Address of a variable that holds the number of initial estimates for the Multiple Imputation method	Optional. Set to $p+p*(p+1)/2$, where p is the task dimension.
VSL_SS_ED_MI_INIT_ESTIMATES	d, s	Address of an array of initial estimates for the Multiple Imputation method	Optional. Set the values of the array according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the <i>Intel® MKL Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_MI_SIMUL_VALS_N	i	Address of a variable that holds the number of simulated values in the Multiple Imputation method	Optional. Positive integer indicating the number of missing points in the observation matrix.
VSL_SS_ED_MI_SIMUL_VALS	d, s	Address of an array of simulated values in the Multiple Imputation method	None.
VSL_SS_ED_MI_ESTIMATES_N	i	Address of a variable that holds the number of estimates obtained as a result of the Multiple Imputation method	Optional. Positive integer number defined according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in

Parameter Value	Type	Purpose	Initialization
			the <i>Intel® MKL Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_MI_ESTIMATES	d, s	Address of an array of estimates obtained as a result of the Multiple Imputation method	None.
VSL_SS_ED_MI_PRIOR_N	i	Address of a variable that holds the number of prior parameters for the Multiple Imputation method	Optional. If you pass a user-defined array of prior parameters, set this parameter to $(p^2+3*p+4)/2$, where p is the task dimension.
VSL_SS_ED_MI_PRIOR	d, s	Address of an array of prior parameters for the Multiple Imputation method	Optional. Set entries of the array of prior parameters according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the <i>Intel® MKL Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_PARAMTR_COR	d, s	Address of a parameterized correlation matrix	None.
VSL_SS_ED_PARAMTR_COR_STORAGE	i	Address of a variable that holds the storage format of a parameterized correlation matrix	Required. Provide a storage format supported by the library whenever you compute the parameterized correlation matrix. ²
VSL_SS_ED_STREAM_QUANT_PARAMS_N	i	Address of a variable that holds the number of parameters of a quantile computation method for streaming data	Required. Set to the number of quantile computation parameters, <i>VSL_SS_SQUANTS_ZW_PARAMS_N</i> .
VSL_SS_ED_STREAM_QUANT_PARAMS	d, s	Address of an array of parameters of a quantile computation method for streaming data	Required. Set the entries of the array according to the description in "Computing Quantiles for Streaming Data" in the <i>Intel® MKL Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_STREAM_QUANT_ORDER_N	i	Address of a variable that holds the number of quantile orders for streaming data	Required. Positive integer value.
VSL_SS_ED_STREAM_QUANT_ORDER	d, s	Address of an array of quantile orders for streaming data	Required. Set entries of the array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_STREAM_QUANT_Q UANTILES	d, s	Address of an array of quantiles for streaming data	None.
VSL_SS_ED_SUM	d, s	Address of array of sums	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_SUM	d, s	Address of array of raw sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_SUM	d, s	Address of array of raw sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_SUM	d, s	Address of array of raw sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_SUM	d, s	Address of array of central sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3C_SUM	d, s	Address of array of central sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4C_SUM	d, s	Address of array of central sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_CP	d, s	Address of cross-product matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_MDAD	d, s	Address of array of median absolute deviations	None.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_MNAD	d, s	Address of array of mean absolute deviations	None.
VSL_SS_ED_SORTED_OBSERV	d, s	Address of the array that stores sorted results	None.
VSL_SS_ED_SORTED_OBSERV_STORAGE	i	Address of a variable that holds the storage format of an output matrix	Required. Provide a supported storage format whenever you specify sorting of the observation matrix.

1. See [Table: "Storage format of matrix of observations and order statistics"](#) for storage formats.
2. See [Table: "Storage formats of a variance-covariance/correlation matrix"](#) for storage formats.

vslSSeditMoments

Modifies the pointers to arrays that hold moment estimates.

Syntax

```
status = vslsseditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
```

```
status = vsldsseditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK)	Descriptor of the task
<i>mean</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of means
<i>r2m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 2 nd order
<i>r3m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 3 rd order
<i>r4m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments REAL(KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of raw moments of the 4 th order

Name	Type	Description
<i>c2m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditmoments</i> REAL(KIND=8) DIMENSION(*) for <i>vsldsseditmoments</i>	Pointer to the array of central moments of the 2 nd order
<i>c3m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditmoments</i> REAL(KIND=8) DIMENSION(*) for <i>vsldsseditmoments</i>	Pointer to the array of central moments of the 3 rd order
<i>c4m</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditmoments</i> REAL(KIND=8) DIMENSION(*) for <i>vsldsseditmoments</i>	Pointer to the array of central moments of the 4 th order

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditMoments` routine replaces pointers to the arrays that hold estimates of raw and central moments with values passed as corresponding parameters of the routine. If an input parameter is `NULL`, the value of the relevant parameter remains unchanged.

vslSSEditSums

Modifies the pointers to arrays that hold sum estimates.

Syntax

```
status = vslsseditsums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
```

```
status = vsldsseditsums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK)	Descriptor of the task
<i>sum</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditsums</i> REAL(KIND=8) DIMENSION(*) for <i>vsldsseditsums</i>	Pointer to the array of sums
<i>r2s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditsums</i>	Pointer to the array of raw sums of the second order

Name	Type	Description
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	
<i>r3s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslSSSeditSums	Pointer to the array of raw sums of the third order
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	
<i>r4s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslSSSeditSums	Pointer to the array of raw sums of the fourth order
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	
<i>c2s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslSSSeditSums	Pointer to the array of central sums of the second order
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	
<i>c3s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslSSSeditSums	Pointer to the array of central sums of the third order
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	
<i>c4s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslSSSeditSums	Pointer to the array of central sums of the fourth order
	REAL(KIND=8) DIMENSION(*) for vslDSSeditSums	

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditSums` routine replaces pointers to the arrays that hold estimates of raw and central sums with values passed as corresponding parameters of the routine. If an input parameter is `NULL`, the value of the relevant parameter remains unchanged.

vslSSEditCovCor

Modifies the pointers to covariance/correlation/cross-product parameters.

Syntax

```
status = vslSSSeditcovcor(task, mean, cov, cov_storage, cor, cor_storage)
```

```
status = vslDSSeditcovcor(task, mean, cov, cov_storage, cor, cor_storage)
```

Include Files

- `mkl_vs1.f90`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>mean</i>	Fortran: REAL (KIND=4) DIMENSION (*) for <i>vslsseditcovcor</i> REAL (KIND=8) DIMENSION (*) for <i>vslseditcovcor</i>	Pointer to the array of means
<i>cov</i>	Fortran: REAL (KIND=4) DIMENSION (*) for <i>vslsseditcovcor</i> REAL (KIND=8) DIMENSION (*) for <i>vslseditcovcor</i>	Pointer to a covariance matrix
<i>cov_storage</i>	Fortran: INTEGER	Pointer to the storage format of the covariance matrix
<i>cor</i>	Fortran: REAL (KIND=4) DIMENSION (*) for <i>vslsseditcovcor</i> REAL (KIND=8) DIMENSION (*) for <i>vslseditcovcor</i>	Pointer to a correlation matrix
<i>cor_storage</i>	Fortran: INTEGER	Pointer to the storage format of the correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslsSEditCovCor` routine replaces pointers to the array of means, covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See [Table "Storage formats of a variance-covariance/correlation/cross-product matrix"](#) for possible values of the *cov_storage* and *cor_storage* parameters. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

Storage formats of a variance-covariance/correlation/cross-product matrix

Parameter	Description
VSL_SS_MATRIX_STORAGE_FULL	A symmetric variance-covariance/correlation/cross-product matrix is a one-dimensional array with elements $c(i, j)$ stored as $cp(i * p + j)$. The size of the array is $p * p$.
VSL_SS_MATRIX_STORAGE_L_PACKED	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + (2n - j) * (j - 1) / 2)$ for $j \leq i$. The size of the array is $p * (p + 1) / 2$.

Parameter	Description
VSL_SS_MATRIX_STORAGE_U_PACKED	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + j*(j - 1)/2)$ for $i \leq j$. The size of the array is $p*(p+ 1)/2$.

vslSSEditCP

Modifies the pointers to cross-product matrix parameters.

Syntax

```
status = vslsseditcp(task, mean, sum, cp, cp_storage)
```

```
status = vsldsseditcp(task, mean, sum, cp, cp_storage)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>mean</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditcp REAL(KIND=8) DIMENSION(*) for vsldsseditcp	Pointer to array of means
<i>sum</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditcp REAL(KIND=8) DIMENSION(*) for vsldsseditcp	Pointer to array of sums
<i>cp</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditcp REAL(KIND=8) DIMENSION(*) for vsldsseditcp	Pointer to a cross-product matrix
<i>cp_storage</i>	Fortran: INTEGER	Pointer to the storage format of the cross-product matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSeditCP` routine replaces pointers to the array of means, array of sums, cross-product matrix, and its storage format with values passed as corresponding parameters of the routine. See [Table: "Storage formats of a variance-covariance/correlation/cross-product matrix"](#) for possible values of the `cp_storage` parameter. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

Storage formats of a variance-covariance/correlation/cross-product matrix

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_FULL</code>	A symmetric variance-covariance/correlation/cross-product matrix is a one-dimensional array with elements $c(i, j)$ stored as $cp(i*p + j)$. The size of the array is $p*p$.
<code>VSL_SS_MATRIX_STORAGE_L_PACKED</code>	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + (2n - j)*(j - 1)/2)$ for $j \leq i$. The size of the array is $p*(p+ 1)/2$.
<code>VSL_SS_MATRIX_STORAGE_U_PACKED</code>	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + j*(j - 1)/2)$ for $i \leq j$. The size of the array is $p*(p+ 1)/2$.

vslSSeditPartialCovCor

Modifies the pointers to partial covariance/correlation parameters.

Syntax

```
status = vslsseditpartialcovcor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage)
```

```
status = vslseditpartialcovcor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE(VSL_SS_TASK)</code>	Descriptor of the task
<code>p_idx_array</code>	Fortran: <code>INTEGER</code>	Pointer to the array that encodes indices of subcomponents Z and Y of the random vector as described in section Mathematical Notation and Definitions . $p_idx_array[i]$ equals to -1 if the i -th component of the random vector belongs to Z

Name	Type	Description
		1, if the i -th component of the random vector belongs to Y .
<i>cov</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a covariance matrix
<i>cov_storage</i>	Fortran: INTEGER	Pointer to the storage format of the covariance matrix
<i>cor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a correlation matrix
<i>cor_storage</i>	Fortran: INTEGER	Pointer to the storage format of the correlation matrix
<i>p_cov</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a partial covariance matrix
<i>p_cov_storage</i>	Fortran: INTEGER	Pointer to the storage format of the partial covariance matrix
<i>p_cor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor	Pointer to a partial correlation matrix
<i>p_cor_storage</i>	Fortran: INTEGER	Pointer to the storage format of the partial correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditPartialCovCor` routine replaces pointers to covariance/correlation arrays, partial covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `cov_storage`, `cor_storage`, `p_cov_storage`, and `p_cor_storage` parameters. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

vslSSEditQuantiles

Modifies the pointers to parameters related to quantile computations.

Syntax

```
status = vslsseditquantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage)
```

```
status = vsldsseditquantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>quant_order_n</i>	Fortran: INTEGER	Pointer to the number of quantile orders
<i>quant_order</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditquantiles	Pointer to the array of quantile orders
<i>quants</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditquantiles	Pointer to the array of quantiles
<i>order_stats</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditquantiles	Pointer to the array of order statistics
<i>order_stats_storage</i>	Fortran: INTEGER	Pointer to the storage format of the order statistics array

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the array that holds order statistics, and the storage format for the order statistics with values passed into the routine. See [Table "Storage format of matrix of observations and order statistics"](#) for possible values of the `order_statistics_storage` parameter. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

vslSSEditStreamQuantiles

Modifies the pointers to parameters related to quantile computations for streaming data.

Syntax

```
status = vslsseditstreamquantiles(task, quant_order_n, quant_order, quants, nparams,
params)
```

```
status = vsldsseditstreamquantiles(task, quant_order_n, quant_order, quants, nparams,
params)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>quant_order_n</i>	Fortran: INTEGER	Pointer to the number of quantile orders
<i>quant_order</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditstreamquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditstreamquantiles	Pointer to the array of quantile orders
<i>quants</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditstreamquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditstreamquantiles	Pointer to the array of quantiles
<i>nparams</i>	Fortran: INTEGER	Pointer to the number of the algorithm parameters
<i>params</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditstreamquantiles REAL (KIND=8) DIMENSION (*) for vsldsseditstreamquantiles	Pointer to the array of the algorithm parameters

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditStreamQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the number of the algorithm parameters, and the array of the algorithm parameters with values passed into the routine. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

vslSSEditPooledCovariance

Modifies pooled/group covariance matrix array pointers.

Syntax

```
status = vslsseditpooledcovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov)
```

```
status = vsldsseditpooledcovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov)
```

Include Files

- mkl_vsl.f90

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>grp_indices</i>	Fortran: INTEGER DIMENSION(*)	Pointer to an array of size <i>n</i> . The <i>i</i> -th element of the array contains the number of the group the observation belongs to.
<i>pld_mean</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditpooledcovariance REAL (KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array of pooled means
<i>pld_cov</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditpooledcovariance REAL (KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array that holds a pooled covariance matrix
<i>req_grp_indices</i>	Fortran: INTEGER DIMENSION(*)	Pointer to the array that contains indices of groups for which estimates to return (such as covariance and mean)
<i>grp_means</i>	Fortran: REAL (KIND=4) DIMENSION(*) for product=Fortran vslsseditpooledcovariance REAL (KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array of group means
<i>grp_cov</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditpooledcovariance REAL (KIND=8) DIMENSION(*) for vsldsseditpooledcovariance	Pointer to the array that holds group covariance matrices

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditPooledCovariance` routine replaces pointers to the array of group indices, the array of pooled means, the array for a pooled covariance matrix, and pointers to the array of indices of group matrices, the array of group means, and the array for group covariance matrices with values passed in the

editors. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged. Use the `vslSSEditTask` routine to replace the storage format for pooled and group covariance matrices.

vslSSEditRobustCovariance

Modifies pointers to arrays related to a robust covariance matrix.

Syntax

```
status = vslsseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
```

```
status = vsldsseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<code>rcov_storage</code>	Fortran: INTEGER	Pointer to the storage format of a robust covariance matrix
<code>nparams</code>	Fortran: INTEGER	Pointer to the number of method parameters
<code>params</code>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditrobustcovariance REAL (KIND=8) DIMENSION (*) for vsldsseditrobustcovariance	Pointer to the array of method parameters
<code>rmean</code>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditrobustcovariance REAL (KIND=8) DIMENSION (*) for vsldsseditrobustcovariance	Pointer to the array of robust means
<code>rcov</code>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditrobustcovariance REAL (KIND=8) DIMENSION (*) for vsldsseditrobustcovariance	Pointer to a robust covariance matrix

Output Parameters

Name	Type	Description
<code>status</code>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditRobustCovariance` routine uses values passed as parameters of the routine to replace:

- pointers to covariance matrix storage
- pointers to the number of method parameters and to the array of the method parameters of size `nparams`
- pointers to the arrays that hold robust means and covariance

See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `rcov_storage` parameter. If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Intel MKL provides a Translated Biweight S-estimator (TBS) for robust estimation of a variance-covariance matrix and mean [[Rocke96](#)]. Use one iteration of the Maronna algorithm with the reweighting step [[Maronna02](#)] to compute the initial point of the algorithm. Pack the parameters of the TBS algorithm into the `params` array and pass them into the editor. [Table "Structure of the Array of TBS Parameters"](#) describes the `params` structure.

Structure of the Array of TBS Parameters

Array Position	Algorithm Parameter	Description
0	ϵ	Breakdown point, the number of outliers the algorithm can hold. By default, the value is $(n-p) / (2n)$.
1	α	Asymptotic rejection probability, see details in [Rocke96]. By default, the value is 0.001.
2	δ	Stopping criterion: the algorithm is terminated if weights are changed less than δ . By default, the value is 0.001.
3	<code>max_iter</code>	Maximum number of iterations. The algorithm terminates after <code>max_iter</code> iterations. By default, the value is 10. If you set this parameter to zero, the function returns a robust estimate of the variance-covariance matrix computed using the Maronna method [Maronna02] only.

The robust estimator of variance-covariance implementation in Intel MKL requires that the number of observations n be greater than twice the number of variables: $n > 2p$.

See additional details of the algorithm usage model in the *Intel® MKL Summary Statistics Application Notes* document [[SS Notes](#)].

`vslSSEditOutliersDetection`

Modifies array pointers related to multivariate outliers detection.

Syntax

```
status = vslsseditoutliersdetection(task, nparams, params, w)
```

```
status = vslseditoutliersdetection(task, nparams, params, w)
```

Include Files

- `mk1_vsl.f90`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>nparams</i>	Fortran: INTEGER	Pointer to the number of method parameters
<i>params</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditoutliersdetection REAL (KIND=8) DIMENSION (*) for vslseditoutliersdetection	Pointer to the array of method parameters
<i>w</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditoutliersdetection REAL (KIND=8) DIMENSION (*) for vslseditoutliersdetection	Pointer to an array of size <i>n</i> . The array holds the weights of observations to be marked as outliers.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslsSEditOutliersDetection` routine uses the parameters passed to replace

- the pointers to the number of method parameters and to the array of the method parameters of size *nparams*
- the pointer to the array that holds the calculated weights of the observations

If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Intel MKL provides the BACON algorithm ([Billor00]) for the detection of multivariate outliers. Pack the parameters of the BACON algorithm into the *params* array and pass them into the editor. [Table "Structure of the Array of BACON Parameters"](#) describes the *params* structure.

Structure of the Array of BACON Parameters

Array Position	Algorithm Parameter	Description
0	Method to start the algorithm	The parameter takes one of the following possible values: VSL_SS_METHOD_BACON_MEDIAN_INIT, if the algorithm is started using the median estimate. This is the default value of the parameter. VSL_SS_METHOD_BACON_MAHALANOBIS_INIT, if the algorithm is started using the Mahalanobis distances.
1	α	One-tailed probability that defines the $(1 - \alpha)$ quantile of χ^2 distribution with <i>p</i> degrees of freedom. The recommended value is α / n , where <i>n</i> is the number of observations. By default, the value is 0.05.

Array Position	Algorithm Parameter	Description
2	δ	Stopping criterion; the algorithm is terminated if the size of the basic subset is changed less than δ . By default, the value is 0.005.

Output of the algorithm is the vector of weights, `BaconWeights`, such that `BaconWeights(i) = 0` if i -th observation is detected as an outlier. Otherwise `BaconWeights(i) = w(i)`, where w is the vector of input weights. If you do not provide the vector of input weights, `BaconWeights(i)` is set to 1 if the i -th observation is not detected as an outlier.

See additional details about usage model of the algorithm in the *Intel(R) MKL Summary Statistics Application Notes* document [[SS Notes](#)].

vsLSSeditMissingValues

Modifies pointers to arrays associated with the method of supporting missing values in a dataset.

Syntax

```
status = vsLSSeditmissingvalues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates)
```

```
status = vsLDSSeditmissingvalues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE(VSL_SS_TASK)</code>	Descriptor of the task
<code>nparams</code>	Fortran: <code>INTEGER</code>	Pointer to the number of method parameters
<code>params</code>	Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> for <code>vsLSSeditmissingvalues</code> <code>REAL(KIND=8) DIMENSION(*)</code> for <code>vsLDSSeditmissingvalues</code>	Pointer to the array of method parameters
<code>init_estimates_n</code>	Fortran: <code>INTEGER</code>	Pointer to the number of initial estimates for mean and a variance-covariance matrix
<code>init_estimates</code>	Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> for <code>vsLSSeditmissingvalues</code> <code>REAL(KIND=8) DIMENSION(*)</code> for <code>vsLDSSeditmissingvalues</code>	Pointer to the array that holds initial estimates for mean and a variance-covariance matrix

Name	Type	Description
<i>prior_n</i>	Fortran: INTEGER	Pointer to the number of prior parameters
<i>prior</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of prior parameters
<i>simul_missing_vals_n</i>	Fortran: INTEGER	Pointer to the size of the array that holds output of the Multiple Imputation method
<i>simul_missing_vals</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of size $k*m$, where k is the total number of missing values, and m is number of copies of missing values. The array holds m sets of simulated missing values for the matrix of observations.
<i>estimates_n</i>	Fortran: INTEGER	Pointer to the number of estimates to be returned by the routine
<i>estimates</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array that holds estimates of the mean and a variance-covariance matrix.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSEditMissingValues` routine uses values passed as parameters of the routine to replace pointers to the number and the array of the method parameters, pointers to the number and the array of initial mean/variance-covariance estimates, the pointer to the number and the array of prior parameters, pointers to the number and the array of simulated missing values, and pointers to the number and the array of the intermediate mean/covariance estimates. If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Before you call the Summary Statistics routines to process missing values, preprocess the dataset and denote missing observations with one of the following predefined constants:

- `VSL_SS_SNAN`, if the dataset is stored in single precision floating-point arithmetic
- `VSL_SS_DNAN`, if the dataset is stored in double precision floating-point arithmetic

Intel MKL provides the `VSL_SS_METHOD_MI` method to support missing values in the dataset based on the Multiple Imputation (MI) approach described in [Schafer97]. The following components support Multiple Imputation:

- Expectation Maximization (EM) algorithm to compute the start point for the Data Augmentation (DA) procedure
- DA function

NOTE

The DA component of the MI procedure is simulation-based and uses the `VSL_BRNG_MCG59` basic random number generator with predefined `seed = 250` and the Gaussian distribution generator (ICDF method) available in Intel MKL [[Gaussian](#)].

Pack the parameters of the MI algorithm into the `params` array. [Table "Structure of the Array of MI Parameters"](#) describes the `params` structure.

Structure of the Array of MI Parameters

Array Position	Algorithm Parameter	Description
0	<code>em_iter_num</code>	Maximal number of iterations for the EM algorithm. By default, this value is 50.
1	<code>da_iter_num</code>	Maximal number of iterations for the DA algorithm. By default, this value is 30.
2	<code>ε</code>	Stopping criterion for the EM algorithm. The algorithm terminates if the maximal module of the element-wise difference between the previous and current parameter values is less than ε . By default, this value is 0.001.
3	<code>m</code>	Number of sets to impute
4	<code>missing_vals_num</code>	Total number of missing values in the datasets

You can also pass initial estimates into the EM algorithm by packing both the vector of means and the variance-covariance matrix as a one-dimensional array `init_estimates`. The size of the array should be at least $p + p(p + 1)/2$. For $i=0, \dots, p-1$, the `init_estimates[i]` array contains the initial estimate of means. The remaining positions of the array are occupied by the upper triangular part of the variance-covariance matrix.

If you provide no initial estimates for the EM algorithm, the editor uses the default values, that is, the vector of zero means and the unitary matrix as a variance-covariance matrix. You can also pass `prior` parameters for μ and Σ into the library: μ_0 , τ , m , and Λ^{-1} . Pack these parameters as a one-dimensional array `prior` with a size of at least

$$(p^2 + 3p + 4)/2.$$

The storage format is as follows:

- `prior[0], \dots, prior[p-1]` contain the elements of the vector μ_0 .
- `prior[p]` contains the parameter τ .
- `prior[p+1]` contains the parameter m .
- The remaining positions are occupied by the upper-triangular part of the inverted matrix Λ^{-1} .

If you provide no `prior` parameters, the editor uses their default values:

- The array of p zeros is used as μ_0 .
- τ is set to 0.
- m is set to p .
- The zero matrix is used as an initial approximate of Λ^{-1} .

The `EditMissingValues` editor returns m sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure.

The editor returns the imputed values as the `simul_missing_vals` array. The size of the array should be sufficient to hold m sets each of the `missing_vals_num` size, that is, at least $m * \text{missing_vals_num}$ in total. The editor packs the imputed values one by one in the order of their appearance in the matrix of observations.

For example, consider a task of dimension 4. The total number of observations n is 10. The second observation vector misses variables 1 and 2, and the seventh observation vector lacks variable 1. The number of sets to impute is $m=2$. Then, `simul_missing_vals[0]` and `simul_missing_vals[1]` contains the first and the second points for the second observation vector, and `simul_missing_vals[2]` holds the first point for the seventh observation. Positions 3, 4, and 5 are formed similarly.

To estimate convergence of the DA algorithm and choose a proper value of the number of DA iterations, request the sequence of parameter estimates that are produced during the DA procedure. The editor returns the sequence of parameters as a single array. The size of the array is

$$m * \text{da_iter_num} * (p + (p^2 + p) / 2)$$

where

- m is the number of sets of values to impute.
- `da_iter_num` is the number of DA iterations.
- The value $p + (p^2 + p) / 2$ determines the size of the memory to hold one set of the parameter estimates.

In each set of the parameters, the vector of means occupies the first p positions and the remaining $(p^2 + p) / 2$ positions are intended for the upper triangular part of the variance-covariance matrix.

Upon successful generation of m sets of imputed values, you can place them in cells of the data matrix with missing values and use the Summary Statistics routines to analyze and get estimates for each of the m complete datasets.

NOTE

Intel MKL implementation of the MI algorithm rewrites cells of the dataset that contain the `VSL_SS_SNAN/VSL_SS_DNAN` values. If you want to use the Summary Statistics routines to process the data with missing values again, mask the positions of the empty cells.

See additional details of the algorithm usage model in the *Intel® MKL Summary Statistics Application Notes* document [[SS Notes](#)].

vslSSEditCorParameterization

Modifies pointers to arrays related to the algorithm of correlation matrix parameterization.

Syntax

```
status = vslsseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
status = vslseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task

Name	Type	Description
<i>cor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditcorparameterization</i> REAL(KIND=8) DIMENSION(*) for <i>vslseditcorparameterization</i>	Pointer to the correlation matrix
<i>cor_storage</i>	Fortran: INTEGER	Pointer to the storage format of the correlation matrix
<i>pcor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <i>vslsseditcorparameterization</i> REAL(KIND=8) DIMENSION(*) for <i>vslseditcorparameterization</i>	Pointer to the parameterized correlation matrix
<i>por_storage</i>	Fortran: INTEGER	Pointer to the storage format of the parameterized correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The *vslSSEditCorParameterization* routine uses values passed as parameters of the routine to replace pointers to the correlation matrix, pointers to the correlation matrix storage format, a pointer to the parameterized correlation matrix, and a pointer to the parameterized correlation matrix storage format. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the *cor_storage* and *pcor_storage* parameters. If an input parameter is *NULL*, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

Task Computation Routines

Task computation routines calculate statistical estimates on the data provided and parameters held in the task descriptor. After you create the task and initialize its parameters, you can call the computation routines as many times as necessary. [Table "Summary Statistics Estimates Obtained with *vslSSCompute* Routine"](#) lists the statistical estimates that you can obtain using the *vslSSCompute* routine.

NOTE

The Summary Statistics computation routines do not signal floating-point errors, such as overflow or gradual underflow, or operations with *NaNs* (except for the missing values in the observations).

Summary Statistics Estimates Obtained with *vslSSCompute* Routine

Estimate	Support of Observations Available in Blocks	Description
<i>VSL_SS_MEAN</i>	Yes	Computes the array of means.
<i>VSL_SS_SUM</i>	Yes	Computes the array of sums.
<i>VSL_SS_2R_MOM</i>	Yes	Computes the array of the 2 nd order raw moments.

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_2R_SUM	Yes	Computes the array of raw sums of the 2 nd order.
VSL_SS_3R_MOM	Yes	Computes the array of the 3 rd order raw moments.
VSL_SS_3R_SUM	Yes	Computes the array of raw sums of the 3 rd order.
VSL_SS_4R_MOM	Yes	Computes the array of the 4 th order raw moments.
VSL_SS_4R_SUM	Yes	Computes the array of raw sums of the 4 th order.
VSL_SS_2C_MOM	Yes	Computes the array of the 2 nd order central moments.
VSL_SS_2C_SUM	Yes	Computes the array of central sums of the 2 nd order.
VSL_SS_3C_MOM	Yes	Computes the array of the 3 rd order central moments.
VSL_SS_3C_SUM	Yes	Computes the array of central sums of the 3 rd order.
VSL_SS_4C_MOM	Yes	Computes the array of the 4 th order central moments.
VSL_SS_4C_SUM	Yes	Computes the array of central sums of the 4 th order.
VSL_SS_KURTOSIS	Yes	Computes the array of kurtosis values.
VSL_SS_SKEWNESS	Yes	Computes the array of skewness values.
VSL_SS_MIN	Yes	Computes the array of minimum values.
VSL_SS_MAX	Yes	Computes the array of maximum values.
VSL_SS_VARIATION	Yes	Computes the array of variation coefficients.
VSL_SS_COV	Yes	Computes a covariance matrix.
VSL_SS_COR	Yes	Computes a correlation matrix. The main diagonal of the correlation matrix holds variances of the random vector components.
VSL_SS_CP	Yes	Computes a cross-product matrix.
VSL_SS_POOLED_COV	No	Computes a pooled covariance matrix.
VSL_SS_POOLED_MEAN	No	Computes an array of pooled means.
VSL_SS_GROUP_COV	No	Computes group covariance matrices.
VSL_SS_GROUP_MEAN	No	Computes group means.
VSL_SS_QUANTS	No	Computes quantiles.
VSL_SS_ORDER_STATS	No	Computes order statistics.

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_ROBUST_COV	No	Computes a robust covariance matrix.
VSL_SS_OUTLIERS	No	Detects outliers in the dataset.
VSL_SS_PARTIAL_COV	No	Computes a partial covariance matrix.
VSL_SS_PARTIAL_COR	No	Computes a partial correlation matrix.
VSL_SS_MISSING_VALS	No	Supports missing values in datasets.
VSL_SS_PARAMTR_COR	No	Computes a parameterized correlation matrix.
VSL_SS_STREAM_QUANTS	Yes	Computes quantiles for streaming data.
VSL_SS_MDAD	No	Computes median absolute deviation.
VSL_SS_MNAD	No	Computes mean absolute deviation.
VSL_SS_SORTED_OBSERV	No	Sorts the dataset by the components of the random vector ξ .

Table "Summary Statistics Computation Method" lists estimate calculation methods supported by Intel MKL. See the *Intel(R) MKL Summary Statistics Application Notes* document [[SS Notes](#)] for a detailed description of the methods.

Summary Statistics Computation Method

Method	Description
VSL_SS_METHOD_FAST	Fast method for calculation of the estimates: <ul style="list-style-type: none"> raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix min/max/quantile/order statistics partial variance-covariance median/mean absolute deviation
VSL_SS_METHOD_FAST_USER_MEAN	Fast method for calculation of the estimates given user-defined mean: <ul style="list-style-type: none"> central moments/sums of 2-4 order, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix, mean absolute deviation
VSL_SS_METHOD_1PASS	One-pass method for calculation of estimates: <ul style="list-style-type: none"> raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix pooled/group covariance matrix
VSL_SS_METHOD_TBS	TBS method for robust estimation of covariance and mean
VSL_SS_METHOD_BACON	BACON method for detection of multivariate outliers
VSL_SS_METHOD_MI	Multiple imputation method for support of missing values

Method	Description
VSL_SS_METHOD_SD	Spectral decomposition method for parameterization of a correlation matrix
VSL_SS_METHOD_SQUANTS_ZW	Zhang-Wang (ZW) method for quantile estimation for streaming data
VSL_SS_METHOD_SQUANTS_ZW_FAST	Fast ZW method for quantile estimation for streaming data
VSL_SS_METHOD_RADIX	Radix method for dataset sorting

You can calculate all requested estimates in one call of the routine. For example, to compute a kurtosis and covariance matrix using a fast method, pass a combination of the pre-defined parameters into the `Compute` routine as shown in the example below:

```
...
method = VSL_SS_METHOD_FAST;
task_params = VSL_SS_KURTOSIS|VSL_SS_COV;
...
status = vsldSSCompute( task, task_params, method );
```

To compute statistical estimates for the next block of observations, you can do one of the following:

- copy the observations to memory, starting with the address available to the task
- use one of the appropriate [Editors](#) to modify the pointer to the new dataset in the task.

The library does not detect your changes of the dataset and computed statistical estimates. To obtain statistical estimates for a new matrix, change the observations and initialize relevant arrays. You can follow this procedure to compute statistical estimates for observations that come in portions. See [Table "Summary Statistics Estimates Obtained with vs1SSCompute Routine"](#) for information on such observations supported by the Intel MKL Summary Statistics estimators.

To modify parameters of the task using the Task Editors, set the address of the targeted matrix of the observations or change the respective vector component indices. After you complete editing the task parameters, you can compute statistical estimates in the modified environment.

If the task completes successfully, the computation routine returns the zero status code. If an error is detected, the computation routine returns an error code. In particular, an error status code is returned in the following cases:

- the task pointer is `NULL`
- memory allocation has failed
- the calculation has failed for some other reason

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vs1SSCompute

Computes Summary Statistics estimates.

Syntax

```
status = vs1sscompute( task, estimates, method)
```

```
status = vsldsscompute( task, estimates, method)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task
<i>estimates</i>	Fortran: INTEGER (KIND=8)	List of statistical estimates to compute
<i>method</i>	Fortran: INTEGER	Method to be used in calculations

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER	Current status of the task

Description

The `vslSSCompute` routine calculates statistical estimates passed as the *estimates* parameter using the algorithms passed as the *method* parameter of the routine. The computations are done in the context of the task descriptor that contains pointers to all required and optional, if necessary, properly initialized arrays. In one call of the function, you can compute several estimates using proper methods for their calculation. See [Table "Summary Statistics Estimates Obtained with Compute Routine"](#) for the list of the estimates that you can calculate with the `vslSSCompute` routine. See [Table "Summary Statistics Computation Methods"](#) for the list of possible values of the *method* parameter.

To initialize single or double precision version task parameters, use the single (`vslsscompute`) or double (`vslDSScompute`) version of the editor, respectively. To initialize parameters of the integer type, use an integer version of the editor (`vslISScompute`).

NOTE

Requesting a combination of the `VSL_SS_MISSING_VALS` value and any other estimate parameter in the `Compute` function results in processing only the missing values.

Application Notes

Be aware that when computing a correlation matrix, the `vslSSCompute` routine allocates an additional array for each thread which is running the task. If you are running on a large number of threads `vslSSCompute` might consume large amounts of memory.

When calculating covariance, correlation, or cross product, the number of bytes of memory required is at least $(P*P*T + P*T)*b$, where P is the dimension of the task or number of variables, T is the number of threads, and b is the number of bytes required for each unit of data. If observation is weighted and the method is `VSL_SS_METHOD_FAST`, then the memory required is at least $(P*P*T + P*T + N*P)*b$, where N is the number of observations.

Task Destructor

Task destructor is the `vslSSDeleteTask` routine intended to delete task objects and release memory.

`vslSSDeleteTask`

Destroys the task object and releases the memory.

Syntax

```
status = vslssdeletetask(task)
```

Include Files

- `mkl_vsl.f90`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: TYPE (VSL_SS_TASK)	Descriptor of the task to destroy

Output Parameters

Name	Type	Description
<code>status</code>	Fortran: INTEGER	Sets to <code>VSL_STATUS_OK</code> if the task is deleted; otherwise a non-zero code is returned.

Description

The `vslSSDeleteTask` routine deletes the task descriptor object, releases the memory allocated for the structure, and sets the task pointer to `NULL`. If `vslSSDeleteTask` fails to delete the task successfully, it returns an error code.

NOTE

Call of the destructor with the `NULL` pointer as the parameter results in termination of the function with no system crash.

Usage Examples

The following examples show various standard operations with Summary Statistics routines.

Calculating Fixed Estimates for Fixed Data

The example shows recurrent calculation of the same estimates with a given set of variables for the complete life cycle of the task in the case of a variance-covariance matrix. The set of vector components to process remains unchanged, and the data comes in blocks. Before you call the `vslSSCompute` routine, initialize pointers to arrays for mean and covariance and set buffers.

```
...
double w[2];
double indices[DIM] = {1, 0, 1};

/* calculating mean for 1st and 3d random vector components */

/* Initialize parameters of the task */
p = DIM;
n = N;

xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

status = vslSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );
```

```
status = vsldSSEditTask ( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );
```

You can process data arrays that come in blocks as follows:

```
for ( i = 0; i < num_of_blocks; i++ )
{
    status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );
    /* Read new data block into array x */
}
...
```

Calculating Different Estimates for Variable Data

The context of your calculation may change in the process of data analysis. The example below shows the data that comes in two blocks. You need to estimate a covariance matrix for the complete data, and the third central moment for the second block of the data using the weights that were accumulated for the previous datasets. The second block of the data is stored in another array. You can proceed as follows:

```
/* Set parameters for the task */
p = DIM;
n = N;
xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

/* Create task */
status = vsldSSNewTask( &task, &p, &n, &xstorage, x1, 0, indices );

/* Initialize the task parameters */
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

/* Calculate covariance for the x1 data */
status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );

/* Initialize array of the 3d central moments and pass the pointer to the task */
for ( i = 0; i < p; i++ ) c3_m[i] = 0.0;

/* Modify task context */
status = vsldSSEditTask( task, VSL_SS_ED_3C_MOM, c3_m );
status = vsldSSEditTask( task, VSL_SS_ED_OBSERV, x2 );

/* Calculate covariance for the x1 & x2 data block */
/* Calculate the 3d central moment for the 2nd data block using earlier accumulated weight */
status = vsldSSCompute( task, VSL_SS_COV|VSL_SS_3C_MOM, VSL_SS_METHOD_FAST );
...
status = vsldSSDeleteTask( &task );
```

Similarly, you can modify indices of the variables to be processed for the next data block.

Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel MKL Summary Statistics functions.

Matrix and Weights of Observations

For a random p -dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, this manual denotes the following:

- $(X)_i = (x_{ij})_{j=1..n}$ is the result of n independent observations for the i -th component ξ_i of the vector ξ .
- The two-dimensional array $X = (x_{ij})_{p \times n}$ is the matrix of observations.
- The column $[X]_j = (x_{ij})_{i=1..p}$ of the matrix X is the j -th observation of the random vector ξ .

Each observation $[X]_j$ is assigned a non-negative weight w_j , where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to n observations of the random vector ξ .
- $$W = \sum_{i=1}^n w_i$$

is the accumulated weight corresponding to observations X .

Vector of sample means

$$M(X) = (M_1(X), \dots, M_p(X)) \text{ with } M_i(X) = \frac{1}{W} \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample partial sums

$$S(X) = (S_1(X), \dots, S_p(X)) \text{ with } S_i(X) = \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample variances

$$V(X) = (V_1(X), \dots, V_p(X)) \text{ with } V_i(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^2, B = W - \sum_{j=1}^n w_j^2 / W$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic moments of k -th order, $k \geq 1$

$$R^{(k)}(X) = (R_1^{(k)}(X), \dots, R_p^{(k)}(X)) \text{ with } R_i^{(k)}(X) = \frac{1}{W} \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic partial sums of k -th order, $k = 2, 3, 4$ (raw/algebraic partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample central moments of the third and the fourth order

$$C^{(k)}(X) = (C_1^{(k)}(X), \dots, C_p^{(k)}(X)) \text{ with } C_i^{(k)}(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^k, B = \sum_{j=1}^n w_j$$

for all $i = 1, \dots, p$ and $k = 3, 4$.

Vector of sample central partial sums of k -th order, $k= 2, 3, 4$ (central partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j (x_{ij} - S_i(X))^k$$

for all $i = 1, \dots, p$.

Vector of sample excess kurtosis values

$$B(X) = (B_1(X), \dots, B_p(X)) \text{ with } B_i(X) = \frac{C_i^{(4)}(X)}{V_i^2(X)} - 3$$

for all $i = 1, \dots, p$.

Vector of sample skewness values

$$\Gamma(X) = (\Gamma_1(X), \dots, \Gamma_p(X)) \text{ with } \Gamma_i(X) = \frac{C_i^{(3)}(X)}{V_i^{1.5}(X)}$$

for all $i = 1, \dots, p$.

Vector of sample variation coefficients

$$VC(X) = (VC_1(X), \dots, VC_p(X)) \text{ with } VC_i(X) = \frac{V_i^{0.5}(X)}{M_i(X)}$$

for all $i = 1, \dots, p$.

Matrix of order statistics

Matrix $Y = (Y_{ij})_{p \times n}$, in which the i -th row $(Y)_i = (Y_{ij})_{j=1 \dots n}$ is obtained as a result of sorting in the ascending order of row $(X)_i = (X_{ij})_{j=1 \dots n}$ in the original matrix of observations.

Vector of sample minimum values

$$Min(X) = (Min_1(X), \dots, Min_p(X)), \text{ where } Min_i(X) = y_{i1}$$

for all $i = 1, \dots, p$.

Vector of sample maximum values

$$Max(X) = (Max_1(X), \dots, Max_p(X)), \text{ where } Max_i(X) = y_{in}$$

for all $i = 1, \dots, p$.

Vector of sample median values

$$Med(X) = (Med_1(X), \dots, Med_p(X)), \text{ where } Med_i(X) = \begin{cases} y_{i,(n+1)/2}, & \text{if } n \text{ is odd} \\ (y_{i,n/2} + y_{i,n/2+1}) / 2, & \text{if } n \text{ is even} \end{cases}$$

for all $i = 1, \dots, p$.

Vector of sample median absolute deviations

$$MDAD(X) = (MDAD_1(X), \dots, MDAD_p(X)), \text{ where } MDAD_i(X) = Med_i(Z) \text{ with } Z = (z_{ij})_{i=1 \dots p, j=1 \dots n}$$

$$z_{ij} = |x_{ij} - Med_i(X)|$$

for all $i = 1, \dots, p$.

Vector of sample mean absolute deviations

$MNAD(X) = (MNAD_1(X), \dots, MNAD_p(X))$, where $MNAD_i(X) = M_i(Z)$ with $Z = (z_{ij})_{i=1\dots p, j=1\dots n}$, $z_{ij} = |x_{ij} - M_i(X)|$ for all $i = 1, \dots, p$.

Vector of sample quantile values

For a positive integer number q and k belonging to the interval $[0, q-1]$, point z_i is the k -th q quantile of the random variable ξ_j if $P\{\xi_i \leq z_i\} \geq \beta$ and $P\{\xi_i \leq z_i\} \geq 1 - \beta$, where

- P is the probability measure.
- $\beta = k/n$ is the quantile order.

The calculation of quantiles is as follows:

$j = [(n-1)\beta]$ and $f = \{(n-1)\beta\}$ as integer and fractional parts of the number $(n-1)\beta$, respectively, and the vector of sample quantile values is

$$Q(X, \beta) = (Q_1(X, \beta), \dots, Q_p(X, \beta))$$

where

$$(Q_i(X, \beta) = Y_{i, j+1} + f(Y_{i, j+2} - Y_{i, j+1}))$$

for all $i = 1, \dots, p$.

Variance-covariance matrix

$$C(X) = (c_{ij}(X))_{p \times p}$$

where

$$c_{ij}(X) = \frac{1}{B} \sum_{k=1}^n w_k (x_{ik} - M_i(X))(x_{jk} - M_j(X)), B = W - \sum_{j=1}^n w_j^2 / W$$

Cross-product matrix (matrix of cross-products and sums of squares)

$$CP(X) = (cp_{ij}(X))_{p \times p}$$

where

$$cp_{ij}(X) = \sum_{k=1}^n w_k (x_{ik} - M_i(X))(x_{jk} - M_j(X))$$

Pooled and group variance-covariance matrices

The set $N = \{1, \dots, n\}$ is partitioned into non-intersecting subsets

$$G_i, i = 1..g, N = \bigcup_{i=1}^g G_i$$

The observation $[X]_j = (x_{ij})_{i=1..p}$ belongs to the group r if $j \in G_r$. One observation belongs to one group only. The group mean and variance-covariance matrices are calculated similarly to the formulas above:

$$M^{(r)}(X) = (M_1^{(r)}(X), \dots, M_p^{(r)}(X)) \text{ with } M_i^{(r)}(X) = \frac{1}{W^{(r)}} \sum_{j \in G_r} w_j x_{ij}, W^{(r)} = \sum_{j \in G_r} w_j$$

for all $i = 1, \dots, p$,

$$C^{(r)}(X) = (c_{ij}^{(r)}(X))_{p \times p}$$

where

$$c_{ij}^{(r)}(X) = \frac{1}{B^{(r)}} \sum_{k \in G_r} w_k (x_{ik} - M_i^{(r)}(X))(x_{jk} - M_j^{(r)}(X)), B^{(r)} = W^{(r)} - \sum_{j \in G_r} w_j^2 / W^{(r)}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

A pooled variance-covariance matrix and a pooled mean are computed as weighted mean over group covariance matrices and group means, correspondingly:

$$M^{pooled}(X) = (M_1^{pooled}(X), \dots, M_p^{pooled}(X)) \text{ with } M_i^{pooled}(X) = \frac{1}{W^{(1)} + \dots + W^{(g)}} \sum_{r=1}^g W^{(r)} M_i^{(r)}(X)$$

for all $i = 1, \dots, p$,

$$C^{pooled}(X) = (c_{ij}^{pooled}(X))_{p \times p}, \quad c_{ij}^{pooled}(X) = \frac{1}{B^{(1)} + \dots + B^{(g)}} \sum_{r=1}^g B^{(r)} c_{ij}^{(r)}(X)$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Correlation matrix

$$R(X) = (r_{ij}(X))_{p \times p}, \text{ where } r_{ij}(X) = \frac{c_{ij}}{\sqrt{c_{ii} c_{jj}}}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Partial variance-covariance matrix

For a random vector ξ partitioned into two components Z and Y , a variance-covariance matrix C describes the structure of dependencies in the vector ξ :

$$C(X) = \begin{pmatrix} C_Z(X) & C_{ZY}(X) \\ C_{YZ}(X) & C_Y(X) \end{pmatrix}.$$

The partial covariance matrix $P(X) = (p_{ij}(X))_{k \times k}$ is defined as

$$P(X) = C_Y(X) - C_{YZ}(X) C_Z^{-1} C_{ZY}(X).$$

where k is the dimension of Y .

Partial correlation matrix

The following is a partial correlation matrix for all $i = 1, \dots, k$ and $j = 1, \dots, k$:

$$RP(X) = (rp_{ij}(X))_{k \times k}, \text{ where } rp_{ij}(X) = \frac{p_{ij}(X)}{\sqrt{p_{ii}(X) p_{jj}(X)}}$$

where

- k is the dimension of Y .
- $p_{ij}(X)$ are elements of the partial variance-covariance matrix.

Sorted dataset

Matrix $Y = (y_{ij})_{p \times n}$, in which the i -th row $(Y)_i$ is obtained as a result of sorting in ascending order the row $(X)_i = (x_{ij})_{j=1..n}$ in the original matrix of observations.

Fourier Transform Functions

The general form of the discrete Fourier transform is

$$z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1 j_2 \dots j_d} \exp\left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l\right)$$

for $k_l = 0, \dots, n_l-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform. In the forward transform, the input (periodic) sequence $\{w_{j_1, j_2, \dots, j_d}\}$ belongs to the set of complex-valued sequences and real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences and complex-valued conjugate-even sequences.

The Intel® Math Kernel Library (Intel® MKL) provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm. Prefixes `Dfti` in function names and `DFTI` in the names of configuration parameters stand for Discrete Fourier Transform Interface.

This chapter describes the following implementations of the fast Fourier transform functions available in Intel MKL:

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see [FFT Functions](#))
- [Cluster FFT functions](#) for distributed-memory architectures (available only for Intel® 64 and Intel® Many Integrated Core architectures)

NOTE

Intel MKL also supports the FFTW3* interfaces to the fast Fourier transform functionality for shared memory paradigm (SMP) systems.

Both FFT and Cluster FFT functions compute an FFT in five steps:

1. Allocate a fresh descriptor for the problem with a call to the `DftiCreateDescriptor` or `DftiCreateDescriptorDM` function. The descriptor captures the configuration of the transform, such as the dimensionality (or rank), sizes, number of transforms, memory layout of the input/output data (defined by strides), and scaling factors. Many of the configuration settings are assigned default values in this call which you might need to modify in your application.
2. Optionally adjust the descriptor configuration with a call to the `DftiSetValue` or `DftiSetValueDM` function as needed. Typically, you must carefully define the data storage layout for an FFT or the data distribution among processes for a Cluster FFT. The configuration settings of the descriptor, such as the default values, can be obtained with the `DftiGetValue` or `DftiGetValueDM` function.
3. Commit the descriptor with a call to the `DftiCommitDescriptor` or `DftiCommitDescriptorDM` function, that is, make the descriptor ready for the transform computation. Once the descriptor is committed, the parameters of the transform, such as the type and number of transforms, strides and distances, the type and storage layout of the data, and so on, are "frozen" in the descriptor.
4. Compute the transform with a call to the `DftiComputeForward/DftiComputeBackward` or `DftiComputeForwardDM/DftiComputeBackwardDM` functions as many times as needed. Because the descriptor is defined and committed separately, all that the compute functions do is take the input and output data and compute the transform as defined. To modify any configuration parameters for another call to a compute function, use `DftiSetValue` followed by `DftiCommitDescriptor` (`DftiSetValueDM` followed by `DftiCommitDescriptorDM`) or create and commit another descriptor.
5. Deallocate the descriptor with a call to the `DftiFreeDescriptor` or `DftiFreeDescriptorDM` function. This returns the memory internally consumed by the descriptor to the operating system.

All the above functions return an integer status value, which is zero upon successful completion of the operation. You can interpret a non-zero status with the help of the [DftiErrorClass](#) or [DftiErrorMessage](#) function.

The FFT functions support lengths with arbitrary factors. You can improve performance of the Intel MKL FFT if the length of your data vector permits factorization into powers of optimized radices. See the *Intel MKL User's Guide* for specific radices supported efficiently.

NOTE

The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel MKL Vector Mathematical Functions provide efficient tools for conversion to and from polar representation (see [Example "Conversion from Cartesian to polar representation of complex data"](#) and [Example "Conversion from polar to Cartesian representation of complex data"](#)).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

FFT Functions

The fast Fourier transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional transforms (of up to seven dimensions) and offers both Fortran and C interfaces for all transform functions.

Table "FFT Functions in Intel MKL" lists FFT functions implemented in Intel MKL:

FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptor	Allocates the descriptor data structure and initializes it with default configuration values.
DftiCommitDescriptor	Performs all initialization for the actual FFT computation.
DftiFreeDescriptor	Frees memory allocated for a descriptor.
DftiCopyDescriptor	Makes a copy of an existing descriptor.
FFT Computation Functions	
DftiComputeForward	Computes the forward FFT.
DftiComputeBackward	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValue	Sets one particular configuration parameter with the specified configuration value.
DftiGetValue	Gets the value of one particular configuration parameter.

Function Name	Operation
Status Checking Functions	
<code>DftiErrorClass</code>	Checks if the status reflects an error of a predefined class.
<code>DftiErrorMessage</code>	Translates the numeric value of an error status into a message.

FFT Interface

The Intel MKL FFT functions are provided with the Fortran and C interfaces. Fortran 95 is required because it offers features that have no counterpart in FORTRAN 77.

NOTE

The Fortran interface of the FFT computation functions requires one-dimensional data arrays for any dimension of FFT problem. For multidimensional transforms, pass the address of the first column of the multidimensional data to the computation functions.

To use the FFT functions, you need to access the module `MKL_DFTI` through the Fortran `use` statement.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`, named constants representing various names of configuration parameters and their possible values, and overloaded functions through the generic functionality of Fortran 95.

NOTE

The current version of the library may not support some of the FFT functions or functionality described in the subsequent sections of this chapter. You can find the complete list of the implementation-specific exceptions in the Intel MKL Release Notes.

For the main categories of Intel MKL FFT functions, see [FFT Functions](#).

Computing an FFT

You can find code examples that compute transforms in the [Fourier Transform Functions Code Examples](#) section in the Code Examples appendix.

Usually you can compute an FFT by five function calls (refer to the [usage model](#) for details). A single data structure, the descriptor, stores configuration parameters that can be changed independently.

The descriptor data structure, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of several configuration parameters. Default settings for some of these parameters are as follows:

- Scale factor: none
- Number of data sets: one
- Data storage: contiguous
- Placement of results: in-place (the computed result overwrites the input data)

The default settings can be changed one at a time through the function `DftiSetValue` as illustrated in [Example "Changing Default Settings \(Fortran\)"](#).

Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module.

All the Intel MKL FFT configuration parameters are readable. Some of them are read-only, while others can be set using the `DftiCreateDescriptor` or `DftiSetValue` function.

Values of the configuration parameters fall into the following groups:

- Values that have native data types. For example, the number of simultaneous transforms requested has an integer value, while the scale factor for a forward transform is a single-precision number.
- Values that are discrete in nature and are provided in the `MKL_DFTI` module as named constants. For example, the domain of the forward transform requires values to be named constants.

Table "Configuration Parameters" summarises the information on configuration parameters, along with their types and values. For more details of each configuration parameter, see the subsection describing this parameter.

Configuration Parameters

Configuration Parameter	Type/Value	Comments
<i>Most common configuration parameters, no default, must be set explicitly by <code>DftiCreateDescriptor</code></i>		
<code>DFTI_PRECISION</code>	Named constant <code>DFTI_SINGLE</code> or <code>DFTI_DOUBLE</code>	Precision of the computation.
<code>DFTI_FORWARD_DOMAIN</code>	Named constant <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code>	Type of the transform.
<code>DFTI_DIMENSION</code>	Integer scalar	Dimension of the transform.
<code>DFTI_LENGTH</code>	Integer scalar/array	Lengths of each dimension.
<i>Common configuration parameters, settable by <code>DftiSetValue</code></i>		
<code>DFTI_PLACEMENT</code>	Named constant <code>DFTI_INPLACE</code> or <code>DFTI_NOT_INPLACE</code>	Defines whether the result overwrites the input data. Default value: <code>DFTI_INPLACE</code> .
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor for the forward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor for the backward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_NUMBER_OF_USER_THREADS</code>	Integer scalar	This configuration parameter is no longer used and kept for compatibility with previous versions of Intel MKL.
<code>DFTI_THREAD_LIMIT</code>	Integer scalar	Limits the number of threads for the DftiComputeForward and DftiComputeBackward . Default value: 0.
<code>DFTI_DESCRIPTOR_NAME</code>	Character string	Assigns a name to a descriptor. Assumed length of the string is <code>DFTI_MAX_NAME_LENGTH</code> . Default value: empty string.
<i>Data layout configuration parameters for single and multiple transforms. Settable by <code>DftiSetValue</code></i>		
<code>DFTI_INPUT_STRIDES</code>	Integer array	Defines the input data layout.

Configuration Parameter	Type/Value	Comments
<code>DFTI_OUTPUT_STRIDES</code>	Integer array	Defines the output data layout.
<code>DFTI_NUMBER_OF_TRANSFORMS</code>	Integer scalar	Number of transforms. Default value: 1.
<code>DFTI_INPUT_DISTANCE</code>	Integer scalar	Defines the distance between input data sets for multiple transforms. Default value: 0.
<code>DFTI_OUTPUT_DISTANCE</code>	Integer scalar	Defines the distance between output data sets for multiple transforms. Default value: 0.
<code>DFTI_COMPLEX_STORAGE</code>	Named constant <code>DFTI_COMPLEX_COMPLEX</code> X or <code>DFTI_REAL_REAL</code>	Defines whether the real and imaginary parts of data for a complex transform are interleaved in one array or split in two arrays. Default value: <code>DFTI_COMPLEX_COMPLEX</code> .
<code>DFTI_REAL_STORAGE</code>	Named constant <code>DFTI_REAL_REAL</code>	Defines how real data for a real transform is stored. Only the <code>DFTI_REAL_REAL</code> value is supported.
<code>DFTI_CONJUGATE_EVEN_STORAGE</code>	Named constant <code>DFTI_COMPLEX_COMPLEX</code> X or <code>DFTI_COMPLEX_REAL</code>	Defines whether the complex data in the backward domain of a real transform is stored as complex elements or as real elements. For the default value, see the detailed description.
<code>DFTI_PACKED_FORMAT</code>	Named constant <code>DFTI_CCE_FORMAT</code> , <code>DFTI_CCS_FORMAT</code> , <code>DFTI_PACK_FORMAT</code> , or <code>DFTI_PERM_FORMAT</code>	Defines the layout of real elements in the backward domain of a one-dimensional or two-dimensional real transform.
<i>Advanced configuration parameters, settable by <code>DftiSetValue</code></i>		
<code>DFTI_WORKSPACE</code>	Named constant <code>DFTI_ALLOW</code> or <code>DFTI_AVOID</code>	Defines whether the library should prefer algorithms using additional memory. Default value: <code>DFTI_ALLOW</code> .
<code>DFTI_ORDERING</code>	Named constant <code>DFTI_ORDERED</code> or <code>DFTI_BACKWARD_SCRAMBLED</code>	Defines whether the result of a complex transform is ordered or permuted. Default value: <code>DFTI_ORDERED</code> .
<i>Read-Only configuration parameters</i>		
<code>DFTI_COMMIT_STATUS</code>	Named constant <code>DFTI_UNCOMMITTED</code> or <code>DFTI_COMMITTED</code>	Readiness of the descriptor for computation.
<code>DFTI_VERSION</code>	String	Version of Intel MKL. Assumed length of the string is <code>DFTI_VERSION_LENGTH</code> .

See Also

[Configuring and Computing an FFT in Fortran](#)

DFTI_PRECISION

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

`DFTI_PRECISION` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

NOTE

Fortran module `MKL_DFTI` also defines named constants `DFTI_SINGLE_R` and `DFTI_DOUBLE_R`, with the same semantics as `DFTI_SINGLE` and `DFTI_DOUBLE`, respectively. Do not use these constants to set the `DFTI_PRECISION` configuration parameter. Use them only as described in section [DftiCreateDescriptor](#).

To better understand configuration of the precision of transforms, you can refer to these examples in your Intel MKL directory:

```
./examples/dftf/source/basic_sp_complex_dft_1d.f90
./examples/dftf/source/basic_dp_complex_dft_1d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)
[DFTI_DIMENSION, DFTI_LENGTHS](#)
[DftiCreateDescriptor](#)

DFTI_FORWARD_DOMAIN

The general form of a discrete Fourier transform is

$$z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1 j_2 \dots j_d} \exp\left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l\right)$$

for $k_l = 0, \dots, n_l-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the backward transform.

The Intel MKL implementation of the FFT algorithm, used for fast computation of discrete Fourier transforms, supports forward transforms on input sequences of two domains, as specified by the `DFTI_FORWARD_DOMAIN` configuration parameter: general complex-valued sequences (`DFTI_COMPLEX` domain) and general real-valued sequences (`DFTI_REAL` domain). The forward transform maps the forward domain to the corresponding backward domain, as shown in [Table "Correspondence of Forward and Backward Domain"](#).

The conjugate-even domain covers complex-valued sequences with the symmetry property:

$$x(k_1, k_2, \dots, k_d) = \text{conjugate}(x(n_1 - k_1, n_2 - k_2, \dots, n_d - k_d))$$

where the index arithmetic is performed modulo respective size, that is,

$$x(\dots, \text{expr}_s, \dots) \equiv x(\dots, \text{mod}(\text{expr}_s, n_s), \dots),$$

and therefore

$$x(\dots, n_s, \dots) \equiv x(\dots, 0, \dots).$$

Due to this property of conjugate-even sequences, only a part of such sequence is stored in the computer memory, as described in [DFTI_CONJUGATE_EVEN_STORAGE](#).

Correspondence of Forward and Backward Domain

Forward Domain	Implied Backward Domain
Complex (DFTI_COMPLEX)	Complex (DFTI_COMPLEX)
Real (DFTI_REAL)	Conjugate-even

DFTI_FORWARD_DOMAIN does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

To better understand usage of the DFTI_FORWARD_DOMAIN configuration parameter, you can refer to these examples in your Intel MKL directory:

```
./examples/dftf/source/basic_sp_complex_dft_1d.f90
./examples/dftf/source/basic_sp_real_dft_1d.f90
```

See Also

[DFTI_PRECISION](#)

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#)

[DftiCreateDescriptor](#)

DFTI_DIMENSION, DFTI_LENGTHS

The dimension of the transform is a positive integer value represented in an integer scalar of `Integer` data type. For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of `Integer` data type. For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer array (of `Integer` data type).

DFTI_DIMENSION and DFTI_LENGTHS do not have a default value. To set them, use the `DftiCreateDescriptor` function and not the `DftiSetValue` function.

To better understand usage of the DFTI_DIMENSION and DFTI_LENGTHS configuration parameters, you can refer to basic examples of one-, two-, and three-dimensional transforms in your Intel MKL directory. Naming conventions for the examples are self-explanatory. For example, refer to these examples of single-precision two-dimensional transforms:

```
./examples/dftf/source/basic_sp_real_dft_2d.f90
./examples/dftf/source/basic_sp_complex_dft_2d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PRECISION](#)

[DftiCreateDescriptor](#)

[DftiSetValue](#)

DFTI_PLACEMENT

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter DFTI_PLACEMENT is DFTI_INPLACE. You can change that by setting it to DFTI_NOT_INPLACE.

NOTE

The data sets have no common elements.

To better understand usage of the DFTI_PLACEMENT configuration parameter, refer to the following example in your Intel MKL directory:

```
./examples/dftf/source/config_placement.f90
```

See Also

DftiSetValue

DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE

The forward transform and backward transform are each associated with a scale factor σ of its own having the default value of 1. You can specify the scale factors using one or both of the configuration parameters `DFTI_FORWARD_SCALE` and `DFTI_BACKWARD_SCALE`. For example, for a one-dimensional transform of length n , you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be $1/n$, thus making the backward transform the inverse of the forward transform.

Set the scale factor configuration parameter using a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

See Also

DftiSetValue

DFTI_PRECISION

DftiGetValue

DFTI_NUMBER_OF_USER_THREADS

The `DFTI_NUMBER_OF_USER_THREADS` configuration parameter is no longer used and kept for compatibility with previous versions of Intel MKL.

See Also

DftiSetValue

DFTI_THREAD_LIMIT

In some situations you may need to limit the number of threads that the `DftiComputeForward` and `DftiComputeBackward` functions use. For example, if more than one thread calls Intel MKL, it might be important that the thread calling these functions does not oversubscribe computing resources (CPU cores). Similarly, a known limit of the maximum number of threads to be used in computations might help the `DftiCommitDescriptor` function to select a more optimal computation method.

Set the parameter `DFTI_THREAD_LIMIT` as follows:

- To a positive number, to specify the maximum number of threads to be used by the compute functions.
- To zero (the default value), to use the maximum number of threads permitted in Intel MKL FFT functions. See "Techniques to Set the Number of Threads" in the *Intel MKL User's Guide* for more information.

On an attempt to set a negative value, the `DftiSetValue` function returns an error and does not update the descriptor.

The value of the `DFTI_THREAD_LIMIT` configuration parameter returned by the `DftiGetValue` function is defined as follows:

- 1 if Intel MKL runs in the sequential mode
- Depends of the commit status of the descriptor if Intel MKL runs in a threaded mode:

Commit Status	Value
Not committed	The value of <code>DFTI_THREAD_LIMIT</code> set in a previous call to the <code>DftiSetValue</code> function or the default value
Committed	The upper limit on the number of threads used by the <code>DftiComputeForward</code> and <code>DftiComputeBackward</code> functions

To better understand usage of the `DFTI_THREAD_LIMIT` configuration parameter, refer to the following example in your Intel MKL directory:

```
./examples/dftf/source/config_thread_limit.f90
```

See Also

- DftiGetValue
- DftiSetValue
- DftiCommitDescriptor
- DftiComputeForward
- DftiComputeBackward
- Threading Control
- DFTI_COMMIT_STATUS

DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For d -dimensional data set X defined by dimensions $N_1 \times N_2 \times \dots \times N_d$, the layout describes where a particular element $X(k_1, k_2, \dots, k_d)$ of the data set is located. The memory address of the element $X(k_1, k_2, \dots, k_d)$ is expressed by the formula

$$\begin{aligned} \text{address of } X(k_1, k_2, \dots, k_d) &= \text{address of } X(0, 0, \dots, 0) + \text{offset} \\ &= \text{address of } X(0, 0, \dots, 0) + s_0 + k_1*s_1 + k_2*s_2 + \dots + k_d*s_d, \end{aligned}$$

where s_0 is the displacement and s_1, \dots, s_d are generalized strides. The configuration parameters `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` enable you to get and set these values. The configuration value is an array of values (s_0, s_1, \dots, s_d) of `INTEGER` data type.

The offset is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the `DFTI_FORWARD_DOMAIN`, `DFTI_COMPLEX_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the type of the elements as shown in [Table "Assumed Element Types of the Input/Output Data"](#):

Assumed Element Types of the Input/Output Data

Descriptor Configuration	Element Type in the Forward Domain	Element Type in the Backward Domain
<code>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</code> <code>DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_COMPLEX</code>	Complex	Complex
<code>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</code> <code>DFTI_COMPLEX_STORAGE=DFTI_REAL_REAL</code>	Real	Real
<code>DFTI_FORWARD_DOMAIN=DFTI_REAL</code> <code>DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL</code>	Real	Real
<code>DFTI_FORWARD_DOMAIN=DFTI_REAL</code> <code>DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX</code>	Real	Complex

The `DFTI_INPUT_STRIDES` configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the `DftiComputeForward` function and by the backward domain for the `DftiComputeBackward` function. The `DFTI_OUTPUT_STRIDES` configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the `DftiComputeForward` function and by the forward domain for `DftiComputeBackward` function.

NOTE

The `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` configuration parameters define the layout of input and output data, and not the forward-domain and backward-domain data. If the data layouts in forward domain and backward domain differ, set `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_STRIDES` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_STRIDES` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

The FFT interface supports both positive and negative stride values. If you use negative strides, set the displacement of the data as follows:

$$s_0 = \sum_{i=1}^d (N_i - 1) \cdot \max(-s_i, 0).$$

The default setting of strides in a general multi-dimensional case assumes that the array that contains the data has no padding. The order of the strides depends on the programming language. For example:

```
INTEGER :: dims(d) = [n1, n2, ..., nd]
status = DftiCreateDescriptor( hand, precision, domain, d, dims)
! The above call assumes data declaration: type X(n1,n2,...,nd)
! Default strides are [ 0, 1, n1, n1*n2, ..., n1*n2*...*nd]
```

Note that in case of a real FFT (`DFTI_FORWARD_DOMAIN=DFTI_REAL`), where different data layouts in the backward domain are available (see [DFTI_PACKED_FORMAT](#)), the default value of the strides is not intuitive for the recommended CCE format (configuration setting `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX`). In case of an *in-place* real transform with the CCE format, set the strides explicitly, as follows:

```
INTEGER :: dims(d) = [n1, n2, ..., nd]
INTEGER :: rstrides(1+d) = [0, 1, 2*(n1/2+1), 2*(n1/2+1)*n2, ... ]
INTEGER :: cstrides(1+d) = [0, 1, (n1/2+1), (n1/2+1)*n2, ... ]
status = DftiCreateDescriptor( hand, precision, domain, d, dims)
status = DftiSetValue( hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
! Set the strides appropriately for forward/backward transform
```

To better understand configuration of strides, you can also refer to these examples in your Intel MKL directory:

```
./examples/dftf/source/basic_sp_real_dft_2d.f90
./examples/dftf/source/basic_sp_real_dft_3d.f90
./examples/dftf/source/basic_dp_real_dft_2d.f90
./examples/dftf/source/basic_dp_real_dft_3d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PLACEMENT](#)

[DftiSetValue](#)

[DftiCommitDescriptor](#)

[DftiComputeForward](#)

[DftiComputeBackward](#)

DFTI_NUMBER_OF_TRANSFORMS

If you need to perform a large number of identical FFTs, you can do this in a single call to a `DftiCompute*` function with the value of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter equal to the actual number of the transforms. The default value of this parameter is one. You can set this parameter to a positive integer value using the `Integer` data type. When setting the number of transforms to a value greater than one, you also need to specify the distance between the input and output data sets using one of the `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` configuration parameters or both.

- The data sets to be transformed must not have common elements.
- All the sets of data must be located within the same memory block.

To better understand usage of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter, refer to the following example in your Intel MKL directory:

```
./examples/dftf/source/config_number_of_transforms.f90
```

See Also

[FFT Computation Functions](#)

[DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE](#)

[DftiSetValue](#)

DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

The FFT interface in Intel MKL enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, `DFTI_INPUT_DISTANCE` for input data or `DFTI_OUTPUT_DISTANCE` for output data, specifies the distribution. The configuration setting is a value of `INTEGER` data type.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see [DFTI_NUMBER_OF_TRANSFORMS](#)).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the `DFTI_FORWARD_DOMAIN`, `DFTI_COMPLEX_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the type of the elements as shown in [Table "Assumed Element Types of the Input/Output Data"](#).

NOTE

The configuration parameters `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` define the distance within input and output data, and not within the forward-domain and backward-domain data. If the distances in the forward and backward domains differ, set `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_DISTANCE` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_DISTANCE` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the data sets on input and output must coincide.

The following example illustrates setting of the `DFTI_INPUT_DISTANCE` configuration parameter:

```
INTEGER :: dims(d) = [n1, n2, ..., nd]
INTEGER :: distance = n1*n2*...*nd
status = DftiCreateDescriptor( hand, precision, DFTI_COMPLEX, d, dims)
status = DftiSetValue( hand, DFTI_NUMBER_OF_TRANSFORMS, howmany )
status = DftiSetValue( hand, DFTI_INPUT_DISTANCE, distance );
```

To better understand configuration of the distances, you can also refer to the following example in your Intel MKL directory:

```
./examples/dftf/source/config_number_of_transforms.f90
```

See Also

[DFTI_PLACEMENT](#)

[DftiSetValue](#)

[DftiCommitDescriptor](#)

DftiComputeForward
DftiComputeBackward

DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE

Depending on the value of the `DFTI_FORWARD_DOMAIN` configuration parameter, the implementation of FFT supports several storage schemes for input and output data (see document [3] for the rationale behind the definition of the storage schemes). The data elements are placed within contiguous memory blocks, defined with generalized strides (see `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see `DFTI_NUMBER_OF_TRANSFORMS` and `DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`).

FFT Examples demonstrate the usage of storage formats.

DFTI_COMPLEX_STORAGE: storage schemes for a complex domain

For the `DFTI_COMPLEX` forward domain, both input and output sequences belong to a complex domain. In this case, the configuration parameter `DFTI_COMPLEX_STORAGE` can have one of the two values: `DFTI_COMPLEX_COMPLEX` (default) or `DFTI_REAL_REAL`.

NOTE

In the Intel MKL FFT interface, storage schemes for a forward complex domain and the respective backward complex domain are the same.

With `DFTI_COMPLEX_COMPLEX` storage, complex-valued data sequences are referenced by a single complex parameter (array) `AZ` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th d -dimensional sequence is located at `AZ[m*distance + stride0 + k1*stride1 + k2*stride2+ ... kd*stride_d]` as a structure consisting of the real and imaginary parts.

The following code illustrates usage of the `DFTI_COMPLEX_COMPLEX` storage:

```
complex :: AZ(N1,N2,N3,M)    ! sizes and number of transforms
...
! on input:  Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)
status = DftiComputeForward( desc, AZ(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)
```

With the `DFTI_REAL_REAL` storage, complex-valued data sequences are referenced by two real parameters `AR` and `AI` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th sequence is computed as `AR[m*distance + stride0 + k1*stride1 + k2*stride2+ ... kd*stride_d] + sqrt(-1) * AI[m*distance + stride0 + k1*stride1 + k2*stride2+ ... kd*stride_d]`.

The following code illustrates usage of the `DFTI_REAL_REAL` storage:

```
real :: AR(N1,N2,N3,M), AI(N1,N2,N3,M)
...
! on input:  Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))
status = DftiComputeForward( desc, AR(:,1,1,1), AI(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))
```

DFTI_REAL_STORAGE: storage schemes for a real domain

The Intel MKL FFT interface supports only one configuration value for this storage scheme: `DFTI_REAL_REAL`. With the `DFTI_REAL_REAL` storage, real-valued data sequences in a real domain are referenced by one real parameter `AR` so that real-valued element of the m -th sequence is located as `AR[m*distance + stride0 + k1*stride1 + k2*stride2+ ... kd*stride_d]`.

DFTI_CONJUGATE_EVEN_STORAGE: storage scheme for a conjugate-even domain

The Intel MKL FFT interface supports two configuration values for this parameter: `DFTI_COMPLEX_REAL` (default) and `DFTI_COMPLEX_COMPLEX`. In both cases, the conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration.

With the `DFTI_COMPLEX_REAL` storage, the complex-valued data sequences in the conjugate-even domain can be reconstructed as described in section `DFTI_PACKED_FORMAT`.

Important

Although `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL` is the default setting for the `DFTI_REAL` forward domain, avoid using this storage scheme because it is supported only for one- and two-dimensional transforms and will be deprecated in future.

With the `DFTI_COMPLEX_COMPLEX` storage, the complex-valued data sequences in the conjugate-even domain are referenced by one complex parameter `AZ` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th sequence can be referenced or reconstructed as described below.

Important

Use the `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX` configuration setting, which will become the default in future. This setting is supported for all transform ranks, provides a uniform pattern for reconstructing the entire conjugate-even sequence from the part of it that is actually stored, and is compatible with data layouts supported by other FFT libraries, such as FFTW. This storage scheme disregards the setting of `DFTI_PACKED_FORMAT`.

Consider a d -dimensional real-to-complex transform

$$Z_{k_1, k_2, \dots, k_d} \equiv \sum_{n_1=0}^{N_1-1} \dots \sum_{n_d=0}^{N_d-1} R_{n_1, n_2, \dots, n_d} e^{\frac{-2\pi i}{N_1} k_1 n_1} \dots e^{\frac{-2\pi i}{N_d} k_d n_d}$$

Because the input sequence R is real-valued, the mathematical result Z has conjugate-even symmetry:

$$z_{k_1, k_2, \dots, k_d} = \text{conjugate}(z_{N_1-k_1, N_2-k_2, \dots, N_d-k_d}),$$

where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued:

$$z_{0, 0, \dots, 0} = \text{conjugate}(z_{0, 0, \dots, 0}).$$

For dimensions with even lengths, the other elements are real-valued too. For example, if N_s is even,

$$z_{0, 0, \dots, N_s/2, 0, \dots, 0} = \text{conjugate}(z_{0, 0, \dots, N_s/2, 0, \dots, 0}).$$

With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension h , it suffices to store elements $z_{k_1, \dots, k_h, \dots, k_d}$ for the following indices:

- $k_h = 0, \dots, [N_h/2]$
- $k_i = 0, \dots, N_i - 1$, where $i = 1, \dots, d$ and $i \neq h$

The symmetry property enables reconstructing the remaining elements: for $k_h = [N_h/2] + 1, \dots, N_h - 1$. In the Intel MKL FFT interface, the halved dimension is the first dimension.

The following code illustrates usage of the `DFTI_COMPLEX_COMPLEX` storage for a conjugate-even domain:

```
real :: AR(N1,N2,M)      ! Array containing values of R
complex :: AZ(N1/2+1,N2,M) ! Array containing values of Z
...
! on input: R{k1,k2,m} = AR(k1,k2,m)
status = DftiComputeForward( desc, AR(:,1,1), AZ(:,1,1) )
! on output:
! for k1=1 ... N1/2+1: Z{k1,k2,m} = AZ(k1,k2,m)
! for k1=N1/2+2 ... N1: Z{k1,k2,m} = conj( AZ(mod(N1-k1+1,N1)+1,mod(N2-k2+1,N2)+1,m) )
```

For the backward transform, the input and output parameters and layouts exchange roles: set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, fwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, bwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeForward( desc, ... )
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, bwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, fwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeBackward( desc, ... )
```

Important

For in-place transforms, ensure the first element of the input data has the same address as the first element of the output data for each dimension.

See Also

[DftiSetValue](#)

DFTI_PACKED_FORMAT

The result of the forward transform of real data is a conjugate-even sequence. Due to the symmetry property, only a part of the complex-valued sequence is stored in memory. The `DFTI_PACKED_FORMAT` configuration parameter defines how the data is packed. Possible values of `DFTI_PACKED_FORMAT` depend on the values of the `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameter:

- `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX`.

The only value of `DFTI_PACKED_FORMAT` can be `DFTI_CCE_FORMAT`. You can use this value with transforms of any dimension. For a description of the corresponding packed format, see [DFTI_CONJUGATE_EVEN_STORAGE](#).

- `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL`.

`DFTI_PACKED_FORMAT` can be `DFTI_CCS_FORMAT`, `DFTI_PACK_FORMAT`, or `DFTI_PERM_FORMAT`. You can use these values with one- and two-dimensional transforms only. The corresponding packed formats are described below.

NOTE

Although `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL` is the default setting for the `DFTI_REAL` forward domain, avoid using this storage scheme because it is supported only for one- and two-dimensional transforms, is incompatible with storage schemes of other FFT libraries, and will be deprecated in future.

DFTI_CCS_FORMAT for One-dimensional Transforms

The following figure illustrates the storage of a one-dimensional (1D) size- N conjugate-even sequence in a real array for the CCS, PACK, and PERM packed formats. The CCS format requires an array of size $N+2$, while the other formats require an array of size N . Zero-based indexing is used.

Storage of a 1D Size- N Conjugate-even Sequence in a Real Array

	n=	0	1	2	3	...	2L-2	2L-1	2L	2L+1	2L+2
CCS, N=2L		R ₀	0	R ₁	I ₁	...	R _{L-1}	I _{L-1}	R _L	0	
PACK, N=2L		R ₀	R ₁	I ₁	R ₂	...	I _{L-1}	R _L			
PERM, N=2L		R ₀	R _L	R ₁	I ₁	...	R _{L-1}	I _{L-1}			
CCS, N=2L+1		R ₀	0	R ₁	I ₁	...	R _{L-1}	I _{L-1}	R _L	I _L	not used
PACK, N=2L+1		R ₀	R ₁	I ₁	R ₂	...	I _{L-1}	R _L	I _L		
PERM, N=2L+1		R ₀	R ₁	I ₁	R ₂	...	I _{L-1}	R _L	I _L		

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array AC as illustrated by figure "Storage of a 1D Size- N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

real :: AR(N), AC(N+2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output:
! for k=1 ... N/2+1: Z{k} = cplx( AC(1 + (2*(k-1)+0)),
!                               AC(1 + (2*(k-1)+1)) )
! for k=N/2+2 ... N: Z{k} = cplx( AC(1 + (2*mod(N-k+1,N)+0)),
!                               -AC(1 + (2*mod(N-k+1,N)+1)))
    
```

DFTI_CCS_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a two-dimensional (2D) M -by- N conjugate-even sequence in a real array for the CCS packed format. This format requires an array of size $(M+2)$ -by- $(N+2)$. Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result. "n/u" means "not used".

Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the CCS Format

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	0
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	0	0
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{1,L}	0
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	I _{1,L}	0
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{K-1,L}	0
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	I _{K-1,L}	0
2K	R _{K,0}	0	n/u	n/u	...	n/u	n/u	R _{K,L}	0
2K+1	0	0	n/u	n/u	...	n/u	n/u	0	0

CCS, M=2K, N=2L

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1	2L+2
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	I _{0,L}	n
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	R _{1,L}	I _{1,L}	n
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{2,L}	I _{2,L}	n
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	R _{3,L}	I _{3,L}	n
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{2K-2,L}	I _{2K-2,L}	n
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	R _{2K-1,L}	I _{2K-1,L}	n
2K	R _{K,0}	0	n/u	n/u	...	n/u	n/u	n/u	n/u	n
2K+1	0	0	n/u	n/u	...	n/u	n/u	n/u	n/u	n

CCS, M=2K, N=2L+1

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	0
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	0	0
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{1,L}	0
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	I _{1,L}	0
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{K-1,L}	0
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	I _{K-1,L}	0
2K	R _{K,0}	0	R _{2K,1}	I _{2K,1}	...	R _{2K,L-1}	I _{2K,L-1}	R _{K,L}	0
2K+1	I _{K,0}	0	n/u	n/u	...	n/u	n/u	I _{K,L}	n/u
2K+2	n/u	n/u	n/u	n/u	...	n/u	n/u	n/u	n/u

CCS, M=2K+1, N=2L

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1	2L+2
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	I _{0,L}	0
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	R _{1,L}	I _{1,L}	0
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{2,L}	I _{2,L}	0
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	R _{3,L}	I _{3,L}	0
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{2K-2,L}	I _{2K-2,L}	0
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	R _{2K-1,L}	I _{2K-1,L}	0
2K	R _{K,0}	0	R _{2K,1}	I _{2K,1}	...	R _{2K,L-1}	I _{2K,L-1}	R _{2K,L}	I _{2K,L}	0
2K+1	I _{K,0}	0	n/u	n/u	...	n/u	n/u	n/u	n/u	0
2K+2	n/u	n/u	n/u	n/u	...	n/u	n/u	n/u	n/u	0

CCS, M=2K+1, N=2L+1

The real and imaginary parts of the complex-valued conjugate-even sequence Z_{k_1, k_2} are located in a real-valued array AC as illustrated by figure "Storage of a 2D M-by-N Conjugate-even Sequence in a Real Array for the CCS Format" and can be used to reconstruct the whole sequence as follows:

```

real :: AR(N1,N2), AC(N1+2,N2+2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 <= N1/2+1) then
!     re = AC(1+2*(k1-1)+0, 1)
!     im = AC(1+2*(k1-1)+1, 1)
!   else
!     re = AC(1+2*(N1-k1+1)+0, 1)
!     im = -AC(1+2*(N1-k1+1)+1, 1)
!   end if
! else if (k1 == 1) then
!   if (k2 <= N2/2+1) then
!     re = AC(1, 1+2*(k2-1)+0)
!     im = AC(1, 1+2*(k2-1)+1)
!   else
!     re = AC(1, 1+2*(N2-k2+1)+0)
!     im = -AC(1, 1+2*(N2-k2+1)+1)
!   end if
! else if (k1-1 == N1-k1+1) then
!   if (k2 <= N2/2+1) then
!     re = AC(N1+1, 1+2*(k2-1)+0)
!     im = AC(N1+1, 1+2*(k2-1)+1)
!   else
!     re = AC(N1+1, 1+2*(N2-k2+1)+0)
!     im = -AC(N1+1, 1+2*(N2-k2+1)+1)
!   end if
! else if (k1 <= N1/2+1) then
!   re = AC(1+2*(k1-1)+0, k2)
!   im = AC(1+2*(k1-1)+1, k2)
! else
!   re = AC(1+2*(N1-k1+1)+0, 1+N2-k2+1)
!   im = -AC(1+2*(N1-k1+1)+1, 1+N2-k2+1)
! end if

```

DFTI_PACK_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array AC as illustrated by figure "Storage of a 1D Size-N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
! if (k == 1) then
!   re = AC(1)
!   im = 0
! else if (k-1 == N-k+1) then
!   re = AC(2*(k-1))
!   im = 0

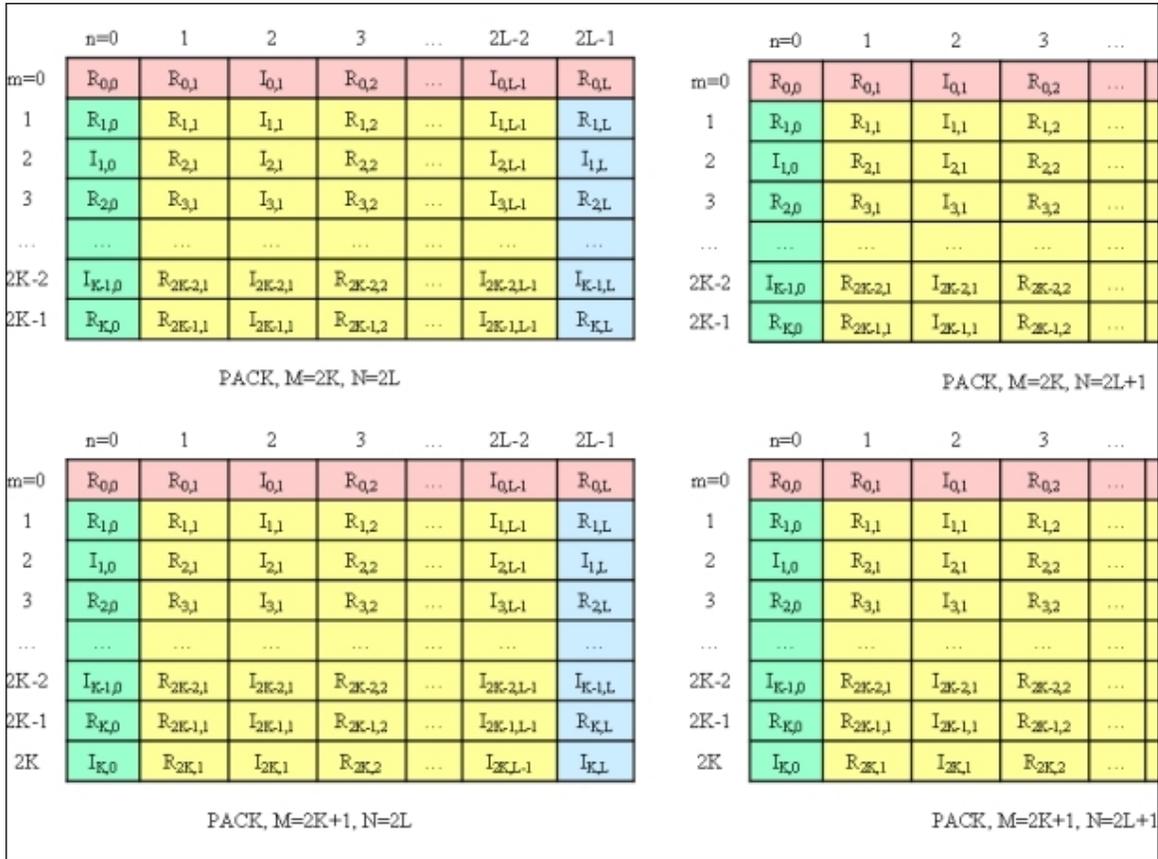
```

```
! else if (k <= N/2+1) then
!   re = AC(2*(k-1)+0)
!   im = AC(2*(k-1)+1)
! else
!   re = AC(2*(N-k+1)+0)
!   im = -AC(2*(N-k+1)+1)
! end if
```

DFTI_PACK_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a 2D M -by- N conjugate-even sequence in a real array for the PACK packed format. This format requires an array of size M -by- N . Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PACK Format



The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in a real-valued array AC as illustrated by figure "Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PACK Format" and can be used to reconstruct the whole sequence as follows:

```
real :: AR(N1,N2), AC(N1,N2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 == 1) then
!     re = AC(1,1)
!     im = 0
```

```

!   else if (k1-1 == N1-k1+1) then
!       re = AC(2*(k1-1),1)
!       im = 0
!   else if (k1 <= N1/2+1) then
!       re = AC(2*(k1-1)+0,1)
!       im = AC(2*(k1-1)+1,1)
!   else
!       re = AC(2*(N1-k1+1)+0,1)
!       im = -AC(2*(N1-k1+1)+1,1)
!   end if
! else if (k1 == 1) then
!   if (k2-1 == N2-k2+1) then
!       re = AC(1,N2)
!       im = 0
!   else if (k2 <= N2/2+1) then
!       re = AC(1,2*(k2-1)+0)
!       im = AC(1,2*(k2-1)+1)
!   else
!       re = AC(1,2*(N2-k2+1)+0)
!       im = -AC(1,2*(N2-k2+1)+1)
!   endif
! else if (k1-1 == N1-k1+1) then
!   if (k2-1 == N2-k2+1) then
!       re = AC(N1,N2)
!       im = 0
!   else if (k2 <= N2/2+1) then
!       re = AC(N1,2*(k2-1)+0)
!       im = AC(N1,2*(k2-1)+1)
!   else
!       re = AC(N1,2*(N2-k2+1)+0)
!       im = -AC(N1,2*(N2-k2+1)+1)
!   end if
! else if (k1 <= N1/2+1) then
!   re = AC(2*(k1-1)+0,1+k2-1)
!   im = AC(2*(k1-1)+1,1+k2-1)
! else
!   re = AC(2*(N1-k1+1)+0,1+N2-k2+1)
!   im = -AC(2*(N1-k1+1)+1,1+N2-k2+1)
! end if

```

DFTI_PERM_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in real-valued array AC as illustrated by figure "Storage of a 1D Size-N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
! if (k == 1) then
!   re = AC(1)
!   im = 0
! else if (k-1 == N-k+1) then
!   re = AC(2)
!   im = 0
! else if (k <= N/2+1) then
!   re = AC(1+2*(k-1)+0-mod(N,2))
!   im = AC(1+2*(k-1)+1-mod(N,2))

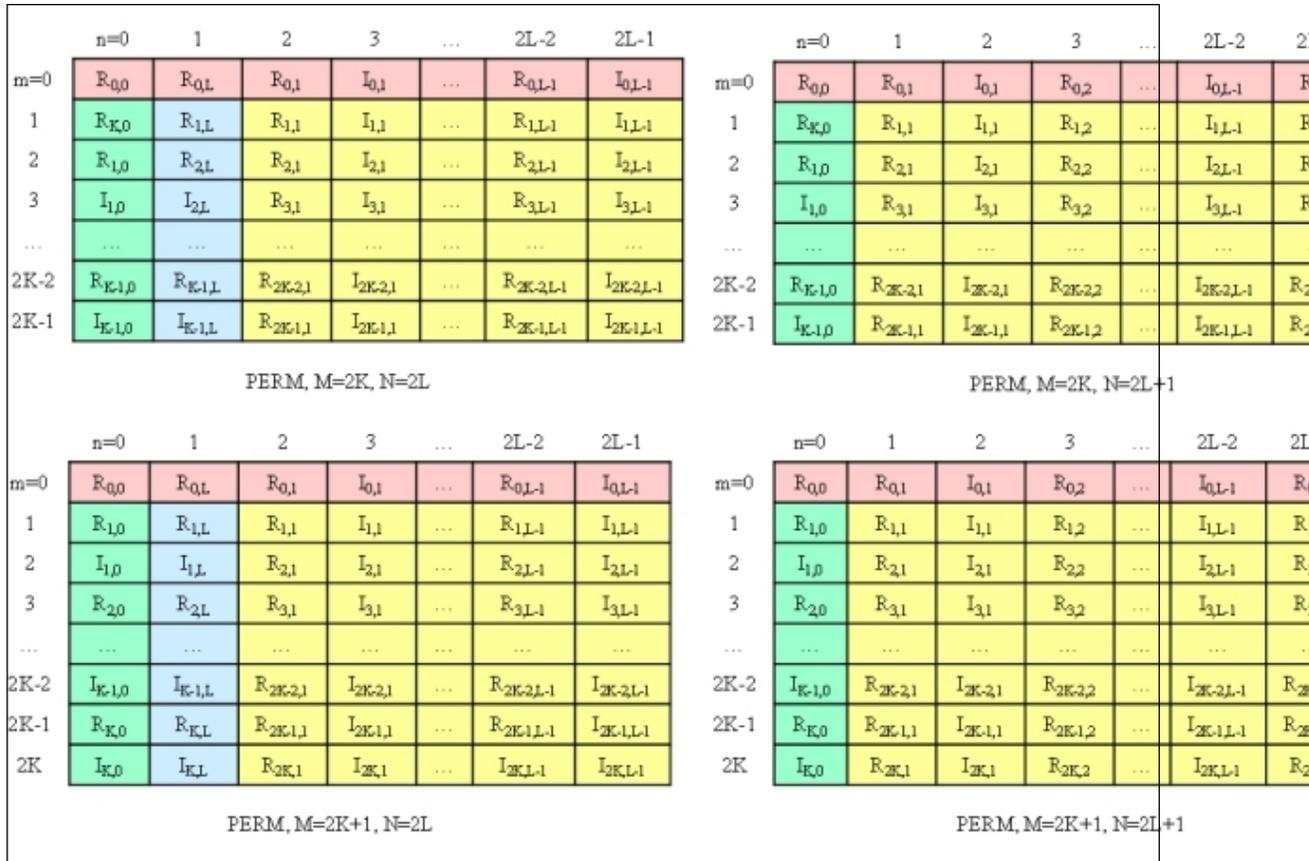
```

```
! else
!   re = AC(1+2*(N-k+1)+0-mod(N,2))
!   im = -AC(1+2*(N-k+1)+1-mod(N,2))
! end if
```

DFTI_PERM_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a 2D M -by- N conjugate-even sequence in a real array for the PERM packed format. This format requires an array of size M -by- N . Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

Storage of a 2D M -by- N Conjugate-Even Sequence in a Real Array for the PERM Format



The real and imaginary parts of the complex-valued conjugate-even sequence Z_{k_1,k_2} are located in real-valued array AC as illustrated by figure "Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PERM Format" and can be used to reconstruct the whole sequence as follows:

```
real :: AR(N1,N2), AC(N1,N2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 == 1) then
!     re = AC(1,1)
!     im = 0
!   else if (k1-1 == N1-k1+1) then
!     re = AC(2,1)
!     im = 0
!   else if (k1 <= N1/2+1) then
```

```

!     re = AC(1+2*(k1-1)+0 - mod(N1,2),1)
!     im = AC(1+2*(k1-1)+1 - mod(N1,2),1)
!     else
!     re = AC(1+2*(N1-k1+1)+0 - mod(N1,2),1)
!     im = -AC(1+2*(N1-k1+1)+1 - mod(N1,2),1)
!     end if
! else if (k1 == 1) then
!     if (k2-1 == N2-k2+1) then
!         re = AC(1,2)
!         im = 0
!     else if (k2 <= N2/2+1) then
!         re = AC(1,1+2*(k2-1)+0 - mod(N2,2))
!         im = AC(1,1+2*(k2-1)+1 - mod(N2,2))
!     else
!         re = AC(1,1+2*(N2-k2+1)+0 - mod(N2,2))
!         im = -AC(1,1+2*(N2-k2+1)+1 - mod(N2,2))
!     endif
! else if (k1-1 == N1-k1+1) then
!     if (k2-1 == N2-k2+1) then
!         re = AC(2,2)
!         im = 0
!     else if (k2 <= N2/2+1) then
!         re = AC(2,1+2*(k2-1)+0-mod(N2,2))
!         im = AC(2,1+2*(k2-1)+1-mod(N2,2))
!     else
!         re = AC(2,1+2*(N2-k2+1)+0-mod(N2,2))
!         im = -AC(2,1+2*(N2-k2+1)+1-mod(N2,2))
!     end if
! else if (k1 <= N1/2+1) then
!     re = AC(1+2*(k1-1)+0-mod(N1,2),1+k2-1)
!     im = AC(1+2*(k1-1)+1-mod(N1,2),1+k2-1)
! else
!     re = AC(1+2*(N1-k1+1)+0-mod(N1,2),1+N2-k2+1)
!     im = -AC(1+2*(N1-k1+1)+1-mod(N1,2),1+N2-k2+1)
! end if

```

To better understand packed formats for two-dimensional transforms, refer to the following example in your Intel MKL directory:

```
./examples/dftf/source/config_conjugate_even_storage.f90
```

See Also

[DftiSetValue](#)

DFTI_WORKSPACE

The computation step for some FFT algorithms requires a scratch space for permutation or other purposes. To manage the use of the auxiliary storage, Intel MKL enables you to set the configuration parameter `DFTI_WORKSPACE` with the following values:

<code>DFTI_ALLOW</code>	(default) Permits the use of the auxiliary storage.
<code>DFTI_AVOID</code>	Instructs Intel MKL to avoid using the auxiliary storage if possible.

See Also

[DftiSetValue](#)

DFTI_COMMIT_STATUS

The `DFTI_COMMIT_STATUS` configuration parameter indicates whether the descriptor is ready for computation. The parameter has two possible values:

<code>DFTI_UNCOMMITTED</code>	Default value, set after a successful call of <code>DftiCreateDescriptor</code> .
<code>DFTI_COMMITTED</code>	The value after a successful call to <code>DftiCommitDescriptor</code> .

A computation function called with an uncommitted descriptor returns an error.

You cannot directly set this configuration parameter in a call to `DftiSetValue`, but a change in the configuration of a committed descriptor may change the commit status of the descriptor to `DFTI_UNCOMMITTED`.

See Also

[DftiCreateDescriptor](#)

[DftiCommitDescriptor](#)

[DftiSetValue](#)

DFTI_ORDERING

Some FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying an FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus using scrambled data is acceptable if it leads to better performance. The following options are available in Intel MKL:

- `DFTI_ORDERED`: Forward transform data ordered, backward transform data ordered (default option).
- `DFTI_BACKWARD_SCRAMBLED`: Forward transform data ordered, backward transform data scrambled.

Table "Scrambled Order Transform" tabulates the effect of this configuration setting.

Scrambled Order Transform

	<code>DftiComputeForward</code>	<code>DftiComputeBackward</code>
DFTI_ORDERING	Input → Output	Input → Output
<code>DFTI_ORDERED</code>	ordered → ordered	ordered → ordered
<code>DFTI_BACKWARD_SCRAMBLED</code>	ordered → scrambled	scrambled → ordered

NOTE

The word "scrambled" in this table means "permit scrambled order if possible". In some situations permitting out-of-order data gives no performance advantage and an implementation may choose to ignore the suggestion.

See Also

[DftiSetValue](#)

Descriptor Manipulation Functions

This category contains the following functions: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

DftiCreateDescriptor

Allocates the descriptor data structure and initializes it with default configuration values.

Syntax

```
status = DftiCreateDescriptor( desc_handle, precision, forward_domain, dimension,  
length )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<i>precision</i>	INTEGER	Precision of the transform: <code>DFTI_SINGLE</code> or <code>DFTI_DOUBLE</code> .
<i>forward_domain</i>	INTEGER	Forward domain of the transform: <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code> .
<i>dimension</i>	INTEGER	Dimension of the transform.
<i>length</i>	INTEGER if <i>dimension</i> = 1. Array INTEGER, DIMENSION(*) otherwise.	Length of the transform for a one-dimensional transform. Lengths of each dimension for a multi-dimensional transform.

Output Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR	FFT descriptor.
<i>status</i>	INTEGER	Function completion status.

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, dimension, and length of the desired transform. Because memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function that can be found in software packages or libraries that implement more traditional algorithms for computing an FFT. This function does not perform any significant computational work such as computation of twiddle factors. The function [DftiCommitDescriptor](#) does this work after the function [DftiSetValue](#) has set values of all necessary parameters.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
```

```
INTERFACE DftiCreateDescriptor

  FUNCTION some_actual_function_1d(desc, precision, domain, dim, length)
    INTEGER :: some_actual_function_1d
    ...
    INTEGER, INTENT(IN) :: length
  END FUNCTION some_actual_function_1d

  FUNCTION some_actual_function_md(desc, precision, domain, dim, lengths)
```

```

    INTEGER :: some_actual_function_md
    ...
    INTEGER, INTENT(IN), DIMENSION(*) :: lengths
END FUNCTION some_actual_function_md

...

END INTERFACE DftiCreateDescriptor

```

Note that the function is overloaded: the actual parameter for the formal parameter *length* can be a scalar or a rank-one array.

The function is also overloaded with respect to the type of the *precision* parameter in order to provide an option of using a precision-specific function for the generic name. Using more specific functions can reduce the size of statically linked executable for the applications using only single-precision FFTs or only double-precision FFTs. To use specific functions, change the "USE MKL_DFTI" statement in your program unit to one of the following:

```

USE MKL_DFTI, FORGET=>DFTI_SINGLE, DFTI_SINGLE=>DFTI_SINGLE_R
USE MKL_DFTI, FORGET=>DFTI_DOUBLE, DFTI_DOUBLE=>DFTI_DOUBLE_R

```

where the name "FORGET" can be replaced with any name that is not used in the program unit.

See Also

[DFTI_PRECISION](#) configuration parameter

[DFTI_FORWARD_DOMAIN](#) configuration parameter

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#) configuration parameters

[Configuration Parameters](#)

DftiCommitDescriptor

Performs all initialization for the actual FFT computation.

Syntax

```
status = DftiCommitDescriptor( desc_handle )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	DFTI_DESCRIPTOR	FFT descriptor.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	DFTI_DESCRIPTOR	Updated FFT descriptor.
<code>status</code>	INTEGER	Function completion status.

Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

If you call the `DftiSetValue` function to change configuration parameters of a committed descriptor (see [Descriptor Configuration Functions](#)), you must re-commit the descriptor before invoking a computation function. Typically, a committal function call is immediately followed by a computation function call (see [FFT Computation Functions](#)).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
FUNCTION some_actual function_1 ( Desc_Handle )
INTEGER :: some_actual function_1
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
END FUNCTION some_actual function_1
END INTERFACE DftiCommitDescriptor
```

DftiFreeDescriptor

Frees the memory allocated for a descriptor.

Syntax

```
status = DftiFreeDescriptor( desc_handle )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DESCRIPTOR_HANDLE</code>	FFT descriptor.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DESCRIPTOR_HANDLE</code>	Memory for the FFT descriptor is released.
<code>status</code>	<code>INTEGER</code>	Function completion status.

Description

This function frees all memory allocated for a descriptor.

NOTE

Memory allocation/deallocation inside Intel MKL is managed by Intel MKL memory management software. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the memory management software sometimes does not return the memory space to the OS, but considers the space free and can reuse it for future memory allocation. See [Example "mkl_free_buffers Usage with FFT Functions"](#) in the description of the service function `FreeBuffers` on how to use Intel MKL memory management software and release memory to the OS.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_3( Desc_Handle )
INTEGER :: some_actual_function_3
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor
```

DftiCopyDescriptor

Makes a copy of an existing descriptor.

Syntax

```
status = DftiCopyDescriptor( desc_handle_original, desc_handle_copy )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<code>desc_handle_original</code>	<code>DESCRIPTOR_HANDLE</code>	The FFT descriptor to copy.

Output Parameters

Name	Type	Description
<code>desc_handle_copy</code>	<code>DESCRIPTOR_HANDLE</code>	The copy of the FFT descriptor.
<code>status</code>	<code>INTEGER</code>	Function completion status.

Description

This function makes a copy of an existing descriptor. The resulting descriptor *desc_handle_copy* and the existing descriptor *desc_handle_original* specify the same configuration of the transform, but do not have any memory areas in common ("deep copy").

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
! argument list and types of dummy arguments. The interface
! does not guarantee what the actual function names are.
! Users can only rely on the function name following the
! keyword INTERFACE
FUNCTION some_actual_function_2( Desc_Handle_Original,
Desc_Handle_Copy )
INTEGER :: some_actual_function_2
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor
```

Descriptor Configuration Functions

This category contains the following functions: the value setting function [DftiSetValue](#) sets one particular configuration parameter to an appropriate value, and the value getting function [DftiGetValue](#) reads the value of one particular configuration parameter. While all configuration parameters are readable, you cannot set a few of them. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, which is derived by the implementation during execution of one of the functions. See [Configuration Settings](#) for details.

DftiSetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

```
status = DftiSetValue( desc_handle, config_param, config_val )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR	FFT descriptor.
<i>config_param</i>	INTEGER	Configuration parameter.
<i>config_val</i>	Depends on the configuration parameter.	Configuration value.

Output Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR	Updated FFT descriptor.
<i>status</i>	INTEGER	Function completion status.

Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- `DFTI_PRECISION`
- `DFTI_FORWARD_DOMAIN`
- `DFTI_DIMENSION`, `DFTI_LENGTH`
- `DFTI_PLACEMENT`
- `DFTI_FORWARD_SCALE`, `DFTI_BACKWARD_SCALE`
- `DFTI_THREAD_LIMIT`
- `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`
- `DFTI_NUMBER_OF_TRANSFORMS`
- `DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`
- `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, `DFTI_CONJUGATE_EVEN_STORAGE`
- `DFTI_PACKED_FORMAT`
- `DFTI_WORKSPACE`
- `DFTI_ORDERING`

The `DftiSetValue` function cannot be used to change configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_PRECISION`, `DFTI_DIMENSION`, and `DFTI_LENGTHS`. Use the `DftiCreateDescriptor` function to set them.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_6_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVAL
END FUNCTION some_actual_function_6_INTVAL
FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_6_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(IN) :: SGLVAL

```

```

END FUNCTION some_actual_function_6_SGLVAL
FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_6_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(OD0)), INTENT(IN) :: DBLVAL
END FUNCTION some_actual_function_6_DBLVAL
FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_6_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVEC(*)
END FUNCTION some_actual_function_6_INTVEC
FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_6_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARACTER(*), INTENT(IN) :: CHARS
END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue

```

See Also

[Configuration Settings](#) for more information on configuration parameters.

[DftiCreateDescriptor](#)

[DftiGetValue](#)

DftiGetValue

Gets the configuration value of one particular configuration parameter.

Syntax

```
status = DftiGetValue( desc_handle, config_param, config_val )
```

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR	FFT descriptor.
<i>config_param</i>	INTEGER	Configuration parameter. See Table "Configuration Parameters" for allowable values of <i>config_param</i> .

Output Parameters

Name	Type	Description
<i>config_val</i>	Depends on the configuration parameter.	Configuration value.
<i>status</i>	INTEGER	Function completion status.

Description

This function gets the configuration value of one particular configuration parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- `DFTI_PRECISION`
- `DFTI_FORWARD_DOMAIN`
- `DFTI_DIMENSION`, `DFTI_LENGTH`
- `DFTI_PLACEMENT`
- `DFTI_FORWARD_SCALE`, `DFTI_BACKWARD_SCALE`
- `DFTI_THREAD_LIMIT`
- `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`
- `DFTI_NUMBER_OF_TRANSFORMS`
- `DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`
- `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, `DFTI_CONJUGATE_EVEN_STORAGE`
- `DFTI_PACKED_FORMAT`
- `DFTI_WORKSPACE`
- `DFTI_COMMIT_STATUS`
- `DFTI_ORDERING`

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

INTERFACE DftiGetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_7_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVAL
END FUNCTION DFTI_GET_VALUE_INTVAL
FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_7_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(OUT) :: SGLVAL
END FUNCTION some_actual_function_7_SGLVAL
FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_7_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
END FUNCTION some_actual_function_7_DBLVAL
FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_7_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVEC(*)
END FUNCTION some_actual_function_7_INTVEC

```

```

FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT )
INTEGER :: some_actual_function_7_INTPNT
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, DIMENSION(*), POINTER :: INTPNT
END FUNCTION some_actual_function_7_INTPNT
FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_7_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARACTER(*), INTENT(OUT):: CHARS
END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue

```

See Also

[Configuration Settings](#) for more information on configuration parameters.
[DftiSetValue](#)

FFT Computation Functions

This category contains the following functions: compute the forward transform and compute the backward transform.

DftiComputeForward

Computes the forward FFT.

Syntax

```

status = DftiComputeForward( desc_handle, x_inout )
status = DftiComputeForward( desc_handle, x_in, y_out )
status = DftiComputeForward( desc_handle, xre_inout, xim_inout )
status = DftiComputeForward( desc_handle, xre_in, xim_in, yre_out, yim_out )

```

Input Parameters

Name	Type	Description
<code>desc_handle</code>	DFTI_DESCRIPTOR	FFT descriptor.
<code>x_inout, x_in</code>	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Data to be transformed in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.
<code>xre_inout, xim_inout, xre_in, xim_in</code>	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- `_inout` to DFTI_INPLACE

- `_in` to `DFTI_NOT_INPLACE`

Output Parameters

Name	Type	Description
<code>y_out</code>	Array <code>REAL(KIND=WP)</code> or <code>COMPLEX(KIND=WP)</code> , <code>DIMENSION(*)</code> , where type and working precision <code>WP</code> must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real backward domain, determined by the <code>DFTI_FORWARD_DOMAIN</code> configuration setting.
<code>xre_inout</code> , <code>xim_inout</code> , <code>yre_out</code> , <code>yim_out</code>	Array <code>REAL(KIND=WP)</code> , <code>DIMENSION(*)</code> , where type and working precision <code>WP</code> must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex backward domain, determined by the <code>DFTI_FORWARD_DOMAIN</code> configuration setting.
<code>status</code>	<code>INTEGER</code>	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_out` to `DFTI_NOT_INPLACE`

Include Files

- `mkl_dfti.f90`

Description

The `DftiComputeForward` function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT, that is, the [transform](#) with the minus sign in the exponent, $\delta = -1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by the generic interface. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_INPUT_STRIDES`, `DFTI_INPUT_DISTANCE`, and so on.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
```

```

INTERFACE DftiComputeForward

FUNCTION some_actual_function_1(desc,sSrcDst)
  INTEGER some_actual_function_1
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
  ...
END FUNCTION some_actual_function_1

FUNCTION some_actual_function_2(desc,cSrcDst)
  INTEGER some_actual_function_2
  COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
  ...
END FUNCTION some_actual_function_2

FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
  INTEGER some_actual_function_3
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
  ...
END FUNCTION some_actual_function_3
...
END INTERFACE DftiComputeForward

```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in [DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the `DIMENSION(*)` attribute and the storage associated with the actual multidimensional arrays via the `EQUIVALENCE` statement.

See Also

[Configuration Settings](#)

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PLACEMENT](#)

[DFTI_PACKED_FORMAT](#)

[DFTI_COMPLEX_STORAGE](#), [DFTI_REAL_STORAGE](#), [DFTI_CONJUGATE_EVEN_STORAGE](#)

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#)

[DFTI_INPUT_DISTANCE](#), [DFTI_OUTPUT_DISTANCE](#)

[DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)

[DftiComputeBackward](#)

[DftiSetValue](#)

DftiComputeBackward

Computes the backward FFT.

Syntax

```
status = DftiComputeBackward( desc_handle, x_inout )
```

```
status = DftiComputeBackward( desc_handle, y_in, x_out )
```

```
status = DftiComputeBackward( desc_handle, xre_inout, xim_inout )
```

```
status = DftiComputeBackward( desc_handle, yre_in, yim_in, xre_out, xim_out )
```

Input Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR	FFT descriptor.
<i>x_inout, y_in</i>	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Data to be transformed in case of a real backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.
<i>xre_inout, xim_inout, yre_in, yim_in</i>	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the data to be transformed in the case of a complex backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_in* to DFTI_NOT_INPLACE

Output Parameters

Name	Type	Description
<i>x_out</i>	Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.
<i>xre_inout, xim_inout, xre_out, xim_out</i>	Array REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.
<i>status</i>	INTEGER	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- *_inout* to DFTI_INPLACE
- *_out* to DFTI_NOT_INPLACE

Include Files

- `mkl_dfti.f90`

Description

The function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, the `DftiComputeBackward` function computes the inverse FFT, that is, the `transform` with the plus sign in the exponent, $\delta = +1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by the generic interface. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_INPUT_STRIDES`, `DFTI_INPUT_DISTANCE`, and so on.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
INTERFACE DftiComputeBackward

  FUNCTION some_actual_function_1(desc,sSrcDst)
    INTEGER some_actual_function_1
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
    ...
  END FUNCTION some_actual_function_1

  FUNCTION some_actual_function_2(desc,cSrcDst)
    INTEGER some_actual_function_2
    COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
    ...
  END FUNCTION some_actual_function_2

  FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
    INTEGER some_actual_function_3
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
    ...
  END FUNCTION some_actual_function_3

  ...
END INTERFACE DftiComputeBackward
```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the `DIMENSION(*)` attribute and the storage associated with the actual multidimensional arrays via the `EQUIVALENCE` statement.

See Also

Configuration Settings

DFTI_FORWARD_DOMAIN
 DFTI_PLACEMENT
 DFTI_PACKED_FORMAT
 DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
 DFTI_DIMENSION, DFTI_LENGTHS
 DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
 DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
 DftiComputeForward
 DftiSetValue

Configuring and Computing an FFT in Fortran

The table below summarizes information on configuring and computing an FFT in Fortran for all kinds of transforms and possible combinations of input and output domains.

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, in-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre>! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_inout) ! or backward FFT status = DftiComputeBackward(hand, X_inout)</pre>
Complex-to-complex, out-of-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre>! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_in, Y_out) ! or backward FFT status = DftiComputeBackward(hand, X_in, Y_out)</pre>
Complex-to-complex, in-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre>! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, Xre_inout, & Xim_inout) ! or backward FFT status = DftiComputeBackward(hand, Xre_inout, & Xim_inout)</pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, out-of-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiSetValue(hand, & DFTI_PLACEMENT, DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, Xre_in, & Xim_in, Yre_out, Yim_out) ! or backward FFT status = DftiComputeBackward(hand, Xre_in, & Xim_in, Yre_out, Yim_out) </pre>
Real-to-complex, in-place, forward	Real numbers	Numbers in the CCE format	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <real_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <complex_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeForward(hand, X_inout) </pre>
Real-to-complex, out-of-place, forward	Real numbers	Numbers in the CCE format	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision> &, DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <real_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <complex_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeForward(hand, X_in, Y_out) </pre>
Complex-to-real, in-place, backward	Numbers in the CCE format	Real numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
			<pre> status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <complex_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <real_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_inout) </pre>
Complex-to-real, out-of-place, backward	Numbers in the CCE format	Real numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <complex_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <real_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_in, Y_out) </pre>

You can find Fortran programs that illustrate configuring and computing FFTs in the `examples/dftf/` subdirectory of your Intel MKL directory.

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. The functions in this category check that status. The first function is a logical function that checks whether the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

DftiErrorClass

Checks whether the status reflects an error of a predefined class.

Syntax

```
predicate = DftiErrorClass( status, error_class )
```

Include Files

- `mkl_dfti.f90`

Input Parameters

Name	Type	Description
<i>status</i>	INTEGER	Completion status of an FFT function.
<i>error_class</i>	INTEGER	Predefined error class.

Output Parameters

Name	Type	Description
<i>predicate</i>	LOGICAL	Result of checking.

Description

The FFT interface in Intel MKL provides a set of predefined error classes listed in [Table "Predefined Error Classes"](#). They are named constants and have the type `INTEGER`.

Predefined Error Classes

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error. The zero status belongs to this class.
<code>DFTI_MEMORY_ERROR</code>	Usually associated with memory allocation
<code>DFTI_INVALID_CONFIGURATION</code>	Invalid settings of one or more configuration parameters
<code>DFTI_INCONSISTENT_CONFIGURATION</code>	Inconsistent configuration or input parameters
<code>DFTI_NUMBER_OF_THREADS_ERROR</code>	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)
<code>DFTI_MULTITHREADED_ERROR</code>	Usually associated with a value that OMP routines return in case of errors
<code>DFTI_BAD_DESCRIPTOR</code>	Descriptor is unusable for computation
<code>DFTI_UNIMPLEMENTED</code>	Unimplemented legitimate settings; implementation dependent
<code>DFTI_MKL_INTERNAL_ERROR</code>	Internal library error
<code>DFTI_1D_LENGTH_EXCEEDS_INT32</code>	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).

The `DftiErrorClass` function returns the value of `.TRUE.` if the status belongs to the predefined error class. To check whether a function call was successful, call `DftiErrorClass` with a specific error class. However, the zero value of the status belongs to the `DFTI_NO_ERROR` class and thus the zero status indicates successful completion of an operation. See [Example "Using Status Checking Functions"](#) for an illustration of correct use of the status checking functions.

NOTE

It is incorrect to directly compare a status with a predefined class.

Interface

```

INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_8( Status, Error_Class )
LOGICAL some_actual_function_8
INTEGER, INTENT(IN) :: Status, Error_Class
END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass

```

DftiErrorMessage

Generates an error message.

Syntax

```
error_message = DftiErrorMessage( status )
```

Include Files

- mkl_dfti.f90

Input Parameters

Name	Type	Description
<i>status</i>	INTEGER	Completion status of a function.

Output Parameters

Name	Type	Description
<i>error_message</i>	CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH)	The character string with the error message.

Description

The error message function generates an error message character string. In Fortran, use a character string of length `DFTI_MAX_MESSAGE_LENGTH` as a target for the error message.

Example "Using Status Checking Function" shows how this function can be used.

Interface

```

INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_9( Status )
CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )

```

```

INTEGER, INTENT(IN) :: Status
END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage

```

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® MKL.

NOTE

These functions are available only for Intel® 64 and Intel® Many Integrated Core architectures.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel MKL provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic skills in MPI programming.

The interfaces for the Intel MKL cluster FFT functions are similar to the corresponding interfaces for the conventional Intel MKL [FFT functions](#), described earlier in this chapter. Refer there for details not explained in this section.

Table "Cluster FFT Functions in Intel MKL" lists cluster FFT functions implemented in Intel MKL:

Cluster FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
<code>DftiCreateDescriptorDM</code>	Allocates memory for the descriptor data structure and preliminarily initializes it.
<code>DftiCommitDescriptorDM</code>	Performs all initialization for the actual FFT computation.
<code>DftiFreeDescriptorDM</code>	Frees memory allocated for a descriptor.
FFT Computation Functions	
<code>DftiComputeForwardDM</code>	Computes the forward FFT.
<code>DftiComputeBackwardDM</code>	Computes the backward FFT.
Descriptor Configuration Functions	
<code>DftiSetValueDM</code>	Sets one particular configuration parameter with the specified configuration value.
<code>DftiGetValueDM</code>	Gets the value of one particular configuration parameter.

Computing Cluster FFT

The cluster FFT functions described later in this section are provided with Fortran and C interfaces. Fortran stands for Fortran 95.

Cluster FFT computation is performed by `DftiComputeForwardDM` and `DftiComputeBackwardDM` functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes are created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of real or complex values. A cluster FFT computation function operates on the local part of the input data, that is, some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel MKL cluster FFT functions are typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_INIT` (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling `DftiCreateDescriptorDM`.
3. Specify one of several values of configuration parameters by one or more calls to `DftiSetValueDM`.
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling `DftiGetValueDM`.
5. Initialize the descriptor for the FFT computation by calling `DftiCommitDescriptorDM`.
6. Create arrays for local parts of input and output data and fill the local part of input data with values. (For more information, see [Distributing Data among Processes](#).)
7. Compute the transform by calling `DftiComputeForwardDM` or `DftiComputeBackwardDM`.
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
9. Release memory allocated for the descriptor by calling `DftiFreeDescriptorDM`.
10. Finalize communication through MPI by calling `MPI_FINALIZE` (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in the "[Examples for Cluster FFT Functions](#)" section in the Code Examples appendix illustrate cluster FFT computations.

Distributing Data among Processes

The Intel MKL cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the column-major order. For example, a two-dimensional matrix A of size (m,n) is stored in a vector B of size $m*n$ so that

$$B((j-1)*m+i) = A(i, j) \quad (i=1, \dots, m, j=1, \dots, n) .$$

NOTE

Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with `Lengths=(m,n,l)` can be computed over an array `AR(m,n,l)`.

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able do this, see sections below on how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C and the last dimension in Fortran. If the global array is two-dimensional, in C, it gives each process several consecutive rows. The term "rows" will be used regardless of the array dimension and programming language. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size $(11,15,12)$, the processes may store local arrays of sizes $(6,15,12)$ and $(5,15,12)$, respectively.

If p is the number of MPI processes and the matrix of a transform to be computed has size (m, n, l) , in C, each MPI process works with local data array of size (m_q, n, l) , where $\sum m_q = m, q = 0, \dots, p-1$. Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain from the following configuration parameters of the cluster FFT interface: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, and `CDFT_LOCAL_SIZE`. To retrieve values of the parameters, use the `DftiGetValueDM` function:

- `CDFT_LOCAL_NX` specifies how many rows of the global array the current process receives.
- `CDFT_LOCAL_START_X` specifies which row of the global input or output array corresponds to the first row of the local input or output array. If A is a global array and L is the appropriate local array, then

$$L(i, j, k) = A(i, j, k + \text{cdft_local_start_x} - 1), \text{ where } i = 1, \dots, m, j = 1, \dots, n, k = 1, \dots, l_q.$$

[Example "2D Out-of-place Cluster FFT Computation"](#) in the [Code Examples appendix](#) shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

One-dimensional transforms

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, *four* configuration parameters are needed: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, `CDFT_LOCAL_OUT_NX`, and `CDFT_LOCAL_OUT_START_X`. Use the `DftiGetValueDM` function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in [Table "Data Distribution Configuration Parameters for 1D Transforms"](#):

Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	<code>CDFT_LOCAL_NX</code>	<code>CDFT_LOCAL_OUT_NX</code>
Elements shift in input array	<code>CDFT_LOCAL_START_X</code>	<code>CDFT_LOCAL_OUT_START_X</code>
Number of elements in output array	<code>CDFT_LOCAL_OUT_NX</code>	<code>CDFT_LOCAL_NX</code>
Elements shift in output array	<code>CDFT_LOCAL_OUT_START_X</code>	<code>CDFT_LOCAL_START_X</code>

Memory size for local data

The memory size needed for local arrays cannot be just calculated from `CDFT_LOCAL_NX` (`CDFT_LOCAL_OUT_NX`), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter `CDFT_LOCAL_SIZE` specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than `CDFT_LOCAL_SIZE * size_of_element`. Note that in the current implementation of the cluster FFT interface, data elements can be real or complex values, each complex value consisting of the real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to [Table "Settable configuration Parameters"](#)), it must have the same size as the local arrays. [Example "1D In-place Cluster FFT Computations"](#) in the [Code Examples appendix](#) illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation performed with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions `MKL_CDFT_ScatterData` and `MKL_CDFT_GatherData` to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with Intel MKL and located in the following subdirectory of the Intel MKL installation directory: `examples/cdftf/source/cdft_example_support.f90`.

Restriction on Lengths of Transforms

The algorithm that the Intel MKL cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, lengths of the last two dimensions must be not less than the number of MPI processes.
- Length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error `CDFT_SPREAD_ERROR` (refer to [Error Codes](#) for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

Cluster FFT Interface

To use the cluster FFT functions, you need to access the module `MKL_CDFT` through the "use" statement.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR_DM`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- Fortran:

```
INCLUDE "mpif.h"
```

(for some MPI versions, "mpif90.h" header may be used instead).

There are three main categories of the cluster FFT functions in Intel MKL:

- 1. Descriptor Manipulation** . There are three functions in this category. The `DftiCreateDescriptorDM` function creates an FFT descriptor whose storage is allocated dynamically. The `DftiCommitDescriptorDM` function "commits" the descriptor to all its settings. The `DftiFreeDescriptorDM` function frees up the memory allocated for the descriptor.
- 2. FFT Computation** . There are two functions in this category. The `DftiComputeForwardDM` function performs the forward FFT computation, and the `DftiComputeBackwardDM` function performs the backward FFT computation.
- 3. Descriptor Configuration** . There are two functions in this category. The `DftiSetValueDM` function sets one specific configuration value to one of the many configuration parameters. The `DftiGetValueDM` function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

DftiCreateDescriptorDM

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, size)
```

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, sizes)
```

Include Files

- `mkl_cdft.f90`

Input Parameters

<i>comm</i>	MPI communicator, e.g. <code>MPI_COMM_WORLD</code> .
<i>v1</i>	Precision of the transform.
<i>v2</i>	Type of the forward domain. Must be <code>DFTI_COMPLEX</code> for complex-to-complex transforms or <code>DFTI_REAL</code> for real-to-complex transforms.
<i>dim</i>	Dimension of the transform.
<i>size</i>	Length of the transform in a one-dimensional case.
<i>sizes</i>	Lengths of the transform in a multi-dimensional case.

Output Parameters

<i>handle</i>	Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.
---------------	---

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" function `DftiCommitDescriptorDM` in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function `DftiSetValueDM`.

The value of the parameter *v1* is specified through named constants `DFTI_SINGLE` and `DFTI_DOUBLE`. It corresponds to precision of input data, output data, and computation. A setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The parameter *dim* is a simple positive integer indicating the dimension of the transform.

In Fortran, length is an integer or an array of integers.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. In this case, the pointer to the created descriptor handle is stored in *handle*. If the function fails, it returns a value of another error class constant

Interface

```
INTERFACE DftiCreateDescriptorDM
  INTEGER(4) FUNCTION DftiCreateDescriptorDMn(C,H,P1,P2,D,L)
```


Include Files

- `mkl_cdft.f90`

Input Parameters

handle The descriptor handle. Must be valid, that is, created in a call to `DftiCreateDescriptorDM`.

Output Parameters

handle The descriptor handle. Memory allocated for the handle is released on output.

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the `DftiFreeDescriptorDM` function to delete the descriptor handle. Upon successful completion of `DftiFreeDescriptorDM` the descriptor handle is no longer valid.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```
INTERFACE DftiFreeDescriptorDM
  INTEGER(4) FUNCTION DftiFreeDescriptorDM(handle)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
  END FUNCTION
END INTERFACE
```

FFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

DftiComputeForwardDM

Computes the forward FFT.

Syntax

```
Status = DftiComputeForwardDM(handle, in_X, out_X)
```

```
Status = DftiComputeForwardDM(handle, in_out_X)
```

Include Files

- `mkl_cdft.f90`

Input Parameters

handle The descriptor handle.

`in_X, in_out_X`

Local part of input data. Array of complex values. Refer to the [Distributing Data among Processes](#) section on how to allocate and initialize the array.

Output Parameters

`out_X, in_out_X`

Local part of output data. Array of complex values. Refer to the [Distributing Data among Processes](#) section on how to allocate the array.

Description

The `DftiComputeForwardDM` function computes the forward FFT. Forward FFT is the transform using the factor $e^{-i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by calling the `DftiComputeForward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeForward`.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data Among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

CAUTION

Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeForwardDM`.

In case of an in-place transform, `DftiComputeForwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

INTERFACE DftiComputeForwardDM
  INTEGER(4) FUNCTION DftiComputeForwardDM(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_x, out_x
  END FUNCTION DftiComputeForwardDM
  INTEGER(4) FUNCTION DftiComputeForwardDMi(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_out_X
  END FUNCTION DftiComputeForwardDMi
  INTEGER(4) FUNCTION DftiComputeForwardDMs(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_x, out_x
  END FUNCTION DftiComputeForwardDMs
  INTEGER(4) FUNCTION DftiComputeForwardDMis(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_out_X
  END FUNCTION DftiComputeForwardDMis
END INTERFACE

```

DftiComputeBackwardDM

Computes the backward FFT.

Syntax

Status = DftiComputeBackwardDM(*handle*, *in_X*, *out_X*)

Status = DftiComputeBackwardDM(*handle*, *in_out_X*)

Include Files

- mkl_cdft.f90

Input Parameters

<i>handle</i>	The descriptor handle.
<i>in_X</i> , <i>in_out_X</i>	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

<i>out_X</i> , <i>in_out_X</i>	Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.
--------------------------------	--

Description

The DftiComputeBackwardDM function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by DftiCreateDescriptorDM, must be configured and committed using the DftiCommitDescriptorDM function.

The computation is carried out by calling the DftiComputeBackward function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of DftiComputeBackward.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

CAUTION

Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeBackwardDM`.

In case of an in-place transform, `DftiComputeBackwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```
INTERFACE DftiComputeBackwardDM
  INTEGER(4) FUNCTION DftiComputeBackwardDM(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_x, out_x
  END FUNCTION DftiComputeBackwardDM
  INTEGER(4) FUNCTION DftiComputeBackwardDMi(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_out_X
  END FUNCTION DftiComputeBackwardDMi
  INTEGER(4) FUNCTION DftiComputeBackwardDMs(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_x, out_x
  END FUNCTION DftiComputeBackwardDMs
  INTEGER(4) FUNCTION DftiComputeBackwardDMis(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_out_X
  END FUNCTION DftiComputeBackwardDMis
END INTERFACE
```

Descriptor Configuration Functions

There are two functions in this category: the value setting function `DftiSetValueDM` sets one particular configuration parameter to an appropriate value, the value getting function `DftiGetValueDM` reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see [Configuration Settings](#) subsection in the "FFT Functions" section for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type `INTEGER(4)`. The exact type of the configuration parameters being floating-point scalars is `REAL(4)` or `REAL(8)`. The configuration parameters whose values are named constants have the `INTEGER` type. They are defined in the `MKL_CDFT` module.

Names of the configuration parameters specific to the cluster FFT interface have prefix `CDFT`.

DftiSetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

```
Status = DftiSetValueDM (handle, param, value)
```

Include Files

- `mkl_cdft.f90`

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to <code>DftiCreateDescriptorDM</code> .
<i>param</i>	Name of a parameter to be set up in the descriptor handle. See Table "Settable Configuration Parameters" for the list of available parameters.
<i>value</i>	Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See [Configuration Settings](#) for details of the meaning of each setting and for possible values of the parameters whose values are named constants.

Settable Configuration Parameters

Parameter Name	Data Type	Description	Default Value
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor of forward transform.	1.0
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor of backward transform.	1.0
<code>DFTI_PLACEMENT</code>	Named constant	Placement of the computation result.	<code>DFTI_INPLACE</code>
<code>DFTI_ORDERING</code>	Named constant	Scrambling of data order.	<code>DFTI_ORDERED</code>

Parameter Name	Data Type	Description	Default Value
CDFT_WORKSPACE	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).
DFTI_PACKED_FORMAT	Named constant	Packed format, real data.	<ul style="list-style-type: none"> DFTI_PERM_FORMAT — default and the only available value for one-dimensional transforms DFTI_CCE_FORMAT — default and the only available value for multi-dimensional transforms
DFTI_TRANSPOSE	Named constant	This parameter determines how the output data is located for multi-dimensional transforms. If the parameter value is DFTI_NONE, the data is located in a usual manner described in this manual. If the value is DFTI_ALLOW, the last (first) global transposition is not performed for a forward (backward) transform.	DFTI_NONE

Return Values

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

INTERFACE DftiSetValueDM
  INTEGER(4) FUNCTION DftiSetValueDM(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p, v
  END FUNCTION
  INTEGER(4) FUNCTION DftiSetValueDMd(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    REAL(8) :: v
  END FUNCTION
  INTEGER(4) FUNCTION DftiSetValueDMs(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    REAL(4) :: v
  END FUNCTION
  INTEGER(4) FUNCTION DftiSetValueDMsw(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    COMPLEX(4) :: v(*)
  END FUNCTION

```

```

INTEGER(4) FUNCTION DftiSetValueMdw(h, p, v)
  TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
  INTEGER(4) :: p
  COMPLEX(8) :: v(*)
END FUNCTION
END INTERFACE

```

DftiGetValueDM

Gets the value of one particular configuration parameter.

Syntax

```
Status = DftiGetValueDM(handle, param, value)
```

Include Files

- mkl_cdft.f90

Input Parameters

handle The descriptor handle. Must be valid, that is, created in a call to [DftiCreateDescriptorDM](#).

param Name of a parameter to be retrieved from the descriptor. See [Table "Retrievable Configuration Parameters"](#) for the list of available parameters.

Output Parameters

value Value of the parameter.

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in [Table "Configuration Parameters"](#) and relevant subsections of the [Configuration Settings](#) section.

Retrievable Configuration Parameters

Parameter Name	Data Type	Description
DFTI_PRECISION	Named constant	Precision of computation, input data and output data.
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.
DFTI_PLACEMENT	Named constant	Placement of the computation result.

Parameter Name	Data Type	Description
DFTI_COMMIT_STATUS	Named constant	Shows whether descriptor has been committed.
DFTI_FORWARD_DOMAIN	Named constant	Forward domain of transforms, has the value of <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code> .
DFTI_ORDERING	Named constant	Scrambling of data order.
CDFT_MPI_COMM	Type of MPI communicator	MPI communicator used for transforms.
CDFT_LOCAL_SIZE	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
CDFT_LOCAL_X_START	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes .
CDFT_LOCAL_NX	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes .
CDFT_LOCAL_OUT_X_START	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes .
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes .

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

INTERFACE DftiGetValueDM
  INTEGER(4) FUNCTION DftiGetValueDM(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p, v
  END FUNCTION
  INTEGER(4) FUNCTION DftiGetValueDMar(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p, v(*)
  END FUNCTION
  INTEGER(4) FUNCTION DftiGetValueDMd(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    REAL(8) :: v
  END FUNCTION
  INTEGER(4) FUNCTION DftiGetValueDMs(h, p, v)

```

```

TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
INTEGER(4) :: p
REAL(4) :: v
END FUNCTION
END INTERFACE

```

Error Codes

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. Named constants specific to the cluster FFT interface have prefix "CDFT" in names. [Table "Error Codes that Cluster FFT Functions Return"](#) lists error codes that the cluster FFT functions may return.

Error Codes that Cluster FFT Functions Return

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error.
<code>DFTI_MEMORY_ERROR</code>	Usually associated with memory allocation.
<code>DFTI_INVALID_CONFIGURATION</code>	Invalid settings of one or more configuration parameters.
<code>DFTI_INCONSISTENT_CONFIGURATION</code>	Inconsistent configuration or input parameters.
<code>DFTI_NUMBER_OF_THREADS_ERROR</code>	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
<code>DFTI_MULTITHREADED_ERROR</code>	Usually associated with a value that OMP routines return in case of errors.
<code>DFTI_BAD_DESCRIPTOR</code>	Descriptor is unusable for computation.
<code>DFTI_UNIMPLEMENTED</code>	Unimplemented legitimate settings; implementation dependent.
<code>DFTI_MKL_INTERNAL_ERROR</code>	Internal library error.
<code>DFTI_1D_LENGTH_EXCEEDS_INT32</code>	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).
<code>CDFT_SPREAD_ERROR</code>	Data cannot be distributed (For more information, see Distributing Data among Processes.)
<code>CDFT_MPI_ERROR</code>	MPI error. Occurs when calling MPI.

PBLAS Routines

This chapter describes the Intel® Math Kernel Library implementation of the PBLAS (Parallel Basic Linear Algebra Subprograms) routines from the ScaLAPACK package for distributed-memory architecture. PBLAS is intended for using in vector-vector, matrix-vector, and matrix-matrix operations to simplify the parallelization of linear codes. The design of PBLAS is as consistent as possible with that of the BLAS. The routine descriptions are arranged in several sections according to the PBLAS level of operation:

- [PBLAS Level 1 Routines](#) (distributed vector-vector operations)
- [PBLAS Level 2 Routines](#) (distributed matrix-vector operations)
- [PBLAS Level 3 Routines](#) (distributed matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by the routine group name; for example, the `p?asum` group, the `p?axpy` group. The question mark in the group name corresponds to a character indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

NOTE

PBLAS routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Generally, PBLAS runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of PBLAS optimized for the target architecture. The Intel MKL version of PBLAS is optimized for Intel® processors. For the detailed system and environment requirements see *Intel® MKL Release Notes* and *Intel® MKL User's Guide*.

For full reference on PBLAS routines and related information, see http://www.netlib.org/scalapack/html/pblas_qref.html.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Overview

The model of the computing environment for PBLAS is represented as a one-dimensional array of processes or also a two-dimensional process grid. To use PBLAS, all global matrices or vectors must be distributed on this array or grid prior to calling the PBLAS routines.

PBLAS uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use PBLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. [Table "Content of the array descriptor for dense matrices"](#) gives an example of an array descriptor structure.

Content of Array Descriptor for Dense Matrices

Array Element #	Name	Definition
1	<i>dtype</i>	Descriptor type (=1 for dense matrices)
2	<i>ctxt</i>	BLACS context handle for the process grid
3	<i>m</i>	Number of rows in the global array
4	<i>n</i>	Number of columns in the global array
5	<i>mb</i>	Row blocking factor
6	<i>nb</i>	Column blocking factor
7	<i>rsrc</i>	Process row over which the first row of the global array is distributed
8	<i>csrc</i>	Process column over which the first column of the global array is distributed
9	<i>lld</i>	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by *LOCr()* and *LOCc()*, respectively. To compute these numbers, you can use the ScaLAPACK tool routine *numroc*.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix *A*, which is contained in the global subarray *sub(A)*, defined by the following 6 values (for dense matrices):

<i>m</i>	The number of rows of <i>sub(A)</i>
<i>n</i>	The number of columns of <i>sub(A)</i>
<i>a</i>	A pointer to the local array containing the entire global array <i>A</i>
<i>ia</i>	The row index of <i>sub(A)</i> in the global array
<i>ja</i>	The column index of <i>sub(A)</i> in the global array
<i>desca</i>	The array descriptor for the global array <i>A</i>

Intel MKL provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (see http://www.netlib.org/scalapack/html/pblas_qref.html).

Routine Naming Conventions

The naming convention for PBLAS routines is similar to that used for BLAS routines (see [Routine Naming Conventions in Chapter 2](#)). A general rule is that each routine name in PBLAS, which has a BLAS equivalent, is simply the BLAS name prefixed by initial letter *p* that stands for "parallel".

The Intel MKL PBLAS routine names have the following structure:

```
p <character> <name> <mod> ( )
```

The *<character>* field indicates the Fortran data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision
i	integer

Some routines and functions can have combined character codes, such as *sc* or *dz*.

For example, the function *pscasum* uses a complex input array and returns a real value.

The *<name>* field, in PBLAS level 1, indicates the operation type. For example, the PBLAS level 1 routines *p?dot*, *p?swap*, *p?copy* compute a vector dot product, vector swap, and a copy vector, respectively.

In PBLAS level 2 and 3, *<name>* reflects the matrix argument type:

<i>ge</i>	general matrix
<i>sy</i>	symmetric matrix
<i>he</i>	Hermitian matrix
<i>tr</i>	triangular matrix

In PBLAS level 3, the *<name>=tran* indicates the transposition of the matrix.

The *<mod>* field, if present, provides additional details of the operation. The PBLAS level 1 names can have the following characters in the *<mod>* field:

<i>c</i>	conjugated vector
<i>u</i>	unconjugated vector

The PBLAS level 2 names can have the following additional characters in the *<mod>* field:

<i>mv</i>	matrix-vector product
<i>sv</i>	solving a system of linear equations with matrix-vector operations
<i>r</i>	rank-1 update of a matrix
<i>r2</i>	rank-2 update of a matrix.

The PBLAS level 3 names can have the following additional characters in the *<mod>* field:

<i>mm</i>	matrix-matrix product
<i>sm</i>	solving a system of linear equations with matrix-matrix operations
<i>rk</i>	rank- <i>k</i> update of a matrix
<i>r2k</i>	rank- <i>2k</i> update of a matrix.

The examples below show how to interpret PBLAS routine names:

<i>pddot</i>	<i><p></i> <i><d></i> <i><dot></i> : double-precision real distributed vector-vector dot product
<i>pcdotc</i>	<i><p></i> <i><c></i> <i><dot></i> <i><c></i> : complex distributed vector-vector dot product, conjugated
<i>pscsum</i>	<i><p></i> <i><sc></i> <i><asum></i> : sum of magnitudes of distributed vector elements, single precision real output and single precision complex input
<i>pcdotu</i>	<i><p></i> <i><c></i> <i><dot></i> <i><u></i> : distributed vector-vector dot product, unconjugated, complex
<i>psgemv</i>	<i><p></i> <i><s></i> <i><ge></i> <i><mv></i> : distributed matrix-vector product, general matrix, single precision
<i>pztrmm</i>	<i><p></i> <i><z></i> <i><tr></i> <i><mm></i> : distributed matrix-matrix product, triangular matrix, double-precision complex.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

PBLAS Level 1 Routines

PBLAS Level 1 includes routines and functions that perform distributed vector-vector operations. [Table "PBLAS Level 1 Routine Groups and Their Data Types"](#) lists the PBLAS Level 1 routine groups and the data types associated with them.

PBLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
<code>p?amax</code>	s, d, c, z	Calculates an index of the distributed vector element with maximum absolute value
<code>p?asum</code>	s, d, sc, dz	Calculates sum of magnitudes of a distributed vector
<code>p?axpy</code>	s, d, c, z	Calculates distributed vector-scalar product
<code>p?copy</code>	s, d, c, z	Copies a distributed vector
<code>p?dot</code>	s, d	Calculates a dot product of two distributed real vectors
<code>p?dotc</code>	c, z	Calculates a dot product of two distributed complex vectors, one of them is conjugated
<code>p?dotu</code>	c, z	Calculates a dot product of two distributed complex vectors
<code>p?nrm2</code>	s, d, sc, dz	Calculates the 2-norm (Euclidean norm) of a distributed vector
<code>p?scal</code>	s, d, c, z, cs, zd	Calculates a product of a distributed vector by a scalar
<code>p?swap</code>	s, d, c, z	Swaps two distributed vectors

`p?amax`

Computes the global index of the element of a distributed vector with maximum absolute value.

Syntax

```
call psamax(n, amax, indx, x, ix, jx, descx, incx)
call pdamax(n, amax, indx, x, ix, jx, descx, incx)
call pcamax(n, amax, indx, x, ix, jx, descx, incx)
call pzamax(n, amax, indx, x, ix, jx, descx, incx)
```

Include Files

- `mkl_pblas.h`

Description

The functions `p?amax` compute global index of the maximum element in absolute value of a distributed vector `sub(x)`,

where `sub(x)` denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$.

Input Parameters

<code>n</code>	(global) INTEGER. The length of distributed vector <code>sub(x)</code> , $n \geq 0$.
<code>x</code>	(local) REAL for <code>psamax</code> DOUBLE PRECISION for <code>pdamax</code> COMPLEX for <code>pcamax</code> DOUBLE COMPLEX for <code>pzamax</code> Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

<code>amax</code>	(global) REAL for <code>psamax</code> . DOUBLE PRECISION for <code>pdamax</code> . COMPLEX for <code>pcamax</code> . DOUBLE COMPLEX for <code>pzamax</code> . Maximum absolute value (magnitude) of elements of the distributed vector only in its scope.
<code>indx</code>	(global) INTEGER. The global index of the maximum element in absolute value of the distributed vector <code>sub(x)</code> only in its scope.

p?asum

Computes the sum of magnitudes of elements of a distributed vector.

Syntax

```
call psasum(n, asum, x, ix, jx, descx, incx)
```

```
call pscasum(n, asum, x, ix, jx, descx, incx)
```

call pdasum(*n*, *asum*, *x*, *ix*, *jx*, *descx*, *incx*)

call pdzasum(*n*, *asum*, *x*, *ix*, *jx*, *descx*, *incx*)

Include Files

- mkl_pblas.h

Description

The functions `p?asum` compute the sum of the magnitudes of elements of a distributed vector `sub(x)`, where `sub(x)` denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vector <code>sub(x)</code> , $n \geq 0$.
<i>x</i>	(local) REAL for <code>psasum</code> DOUBLE PRECISION for <code>pdasum</code> COMPLEX for <code>pscasum</code> DOUBLE COMPLEX for <code>pdzasum</code> Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>asum</i>	(local) REAL for <code>psasum</code> and <code>pscasum</code> . DOUBLE PRECISION for <code>pdasum</code> and <code>pdzasum</code> Contains the sum of magnitudes of elements of the distributed vector only in its scope.
-------------	---

p?axpy

Computes a distributed vector-scalar product and adds the result to a distributed vector.

Syntax

call psaxpy(*n*, *a*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*)

call pdaxpy(*n*, *a*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*)

call pcaxpy(*n*, *a*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*)

call pzaxpy(*n*, *a*, *x*, *ix*, *jx*, *descx*, *incx*, *y*, *iy*, *jy*, *descy*, *incy*)

Include Files

- mkl_pblas.h

Description

The `p?axpy` routines perform the following operation with distributed vectors:

```
sub(y) := sub(y) + a*sub(x)
```

where:

a is a scalar;

`sub(x)` and `sub(y)` are n -element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy:iy+n-1, jy)$ if $incy=1$.

Input Parameters

n	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
a	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Specifies the scalar a .
x	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Array, size $(jx-1)*m_x + ix + (n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
ix, jx	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
$descx$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
y	(local) REAL for psaxpy DOUBLE PRECISION for pdaxpy COMPLEX for pcaxpy DOUBLE COMPLEX for pzaxpy Array, size $(jy-1)*m_y + iy + (n-1)*abs(incy)$.

This array contains the entries of the distributed vector $\text{sub}(y)$.

<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

y Overwritten by $\text{sub}(y) := \text{sub}(y) + a * \text{sub}(x)$.

p?copy

Copies one distributed vector to another vector.

Syntax

```
call pscopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pccopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call picopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?copy routines perform a copy operation with distributed vectors defined as

$\text{sub}(y) = \text{sub}(x)$,

where $\text{sub}(x)$ and $\text{sub}(y)$ are n -element distributed vectors.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$;

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy:iy+n-1, jy)$ if $incy=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) REAL for pscopy DOUBLE PRECISION for pdcopy COMPLEX for pccopy DOUBLE COMPLEX for pzcopy INTEGER for picopy

	Array, size $(j_x-1)*m_x + ix+(n-1)*abs(incx)$.
	This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix <code>X</code> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <code>X</code> .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <code>incx</code> must not be zero.
<code>y</code>	(local) REAL for <code>pscopy</code> DOUBLE PRECISION for <code>pdcopy</code> COMPLEX for <code>pccopy</code> DOUBLE COMPLEX for <code>pzcopy</code> INTEGER for <code>picopy</code> Array, size $(j_y-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<code>iy, jy</code>	(global) INTEGER. The row and column indices in the distributed matrix <code>Y</code> indicating the first row and the first column of the submatrix <code>sub(Y)</code> , respectively.
<code>descy</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <code>Y</code> .
<code>incy</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and m_y . <code>incy</code> must not be zero.

Output Parameters

`y` Overwritten with the distributed vector `sub(x)`.

p?dot

Computes the dot product of two distributed real vectors.

Syntax

```
call psdot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

```
call pddot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- `mkl_pblas.h`

Description

The `?dot` functions compute the dot product *dot* of two distributed real vectors defined as

$$dot = sub(x) \cdot sub(y)$$

where `sub(x)` and `sub(y)` are *n*-element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy:iy+n-1, jy)$ if $incy=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) REAL for <code>psdot</code> DOUBLE PRECISION for <code>pddot</code> Array, size $(jx-1) * m_x + ix + (n-1) * abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) REAL for <code>psdot</code> DOUBLE PRECISION for <code>pddot</code> Array, size $(jy-1) * m_y + iy + (n-1) * abs(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <code>sub(Y)</code> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>dot</i>	(local) REAL for <code>psdot</code> DOUBLE PRECISION for <code>pddot</code> Dot product of <code>sub(x)</code> and <code>sub(y)</code> only in their scope.
------------	---

p?dotc

Computes the dot product of two distributed complex vectors, one of them is conjugated.

Syntax

```
call pcdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

```
call pzdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?dotc functions compute the dot product *dotc* of two distributed vectors, with one vector conjugated:

$$dotc = \text{conjg}(\text{sub}(x)') * \text{sub}(y)$$

where $\text{sub}(x)$ and $\text{sub}(y)$ are n -element distributed vectors.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$;

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy:iy+n-1, jy)$ if $incy=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) COMPLEX for pcdotc DOUBLE COMPLEX for pzdotc Array, size $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for pcdotc DOUBLE COMPLEX for pzdotc Array, size $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.

descy (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *Y*.

incy (global) INTEGER. Specifies the increment for the elements of *sub(y)*. Only two values are supported, namely 1 and *m_y*. *incy* must not be zero.

Output Parameters

dotc (local) COMPLEX for *pcdotc*
 DOUBLE COMPLEX for *pzdotc*
 Dot product of *sub(x)* and *sub(y)* only in their scope.

p?dotu

Computes the dot product of two distributed complex vectors.

Syntax

```
call pcdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- `mkl_pblas.h`

Description

The *p?dotu* functions compute the dot product *dotu* of two distributed vectors defined as

```
dotu = sub(x)'*sub(y)
```

where *sub(x)* and *sub(y)* are *n*-element distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if *incx*=*m_x*, and $X(ix: ix+n-1, jx)$ if *incx*= 1;

sub(y) denotes $Y(iy, jy:jy+n-1)$ if *incy*=*m_y*, and $Y(iy: iy+n-1, jy)$ if *incy*= 1.

Input Parameters

n (global) INTEGER. The length of distributed vectors, $n \geq 0$.

x (local) COMPLEX for *pcdotu*
 DOUBLE COMPLEX for *pzdotu*
 Array, size $(jx-1)*m_x + ix + (n-1)*abs(incx)$.
 This array contains the entries of the distributed vector *sub(x)*.

ix, jx (global) INTEGER. The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix *sub(X)*, respectively.

descx (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *X*.

<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for <code>pcdotu</code> DOUBLE COMPLEX for <code>pzdotu</code> Array, size $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>dotu</i>	(local) COMPLEX for <code>pcdotu</code> DOUBLE COMPLEX for <code>pzdotu</code> Dot product of $\text{sub}(x)$ and $\text{sub}(y)$ only in their scope.
-------------	--

p?nrm2

Computes the Euclidean norm of a distributed vector.

Syntax

```
call psnrm2(n, norm2, x, ix, jx, descx, incx)
call pdnrm2(n, norm2, x, ix, jx, descx, incx)
call pscnrm2(n, norm2, x, ix, jx, descx, incx)
call pdznm2(n, norm2, x, ix, jx, descx, incx)
```

Include Files

- `mkl_pblas.h`

Description

The `p?nrm2` functions compute the Euclidean norm of a distributed vector $\text{sub}(x)$, where $\text{sub}(x)$ is an n -element distributed vector.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
<i>x</i>	(local) REAL for <code>psnrm2</code>

DOUBLE PRECISION for `pdnrm2`

COMPLEX for `pscnrm2`

DOUBLE COMPLEX for `pdznrm2`

Array, size $(j_x-1)*m_x + ix+(n-1)*abs(incx)$.

This array contains the entries of the distributed vector `sub(x)`.

`ix, jx`

(global) INTEGER. The row and column indices in the distributed matrix `X` indicating the first row and the first column of the submatrix `sub(X)`, respectively.

`descx`

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix `X`.

`incx`

(global) INTEGER. Specifies the increment for the elements of `sub(x)`. Only two values are supported, namely 1 and m_x . `incx` must not be zero.

Output Parameters

`norm2`

(local) REAL for `psnrm2` and `pscnrm2`.

DOUBLE PRECISION for `pdnrm2` and `pdznrm2`

Contains the Euclidean norm of a distributed vector only in its scope.

p?scal

Computes a product of a distributed vector by a scalar.

Syntax

```
call psscal(n, a, x, ix, jx, descx, incx)
call pdscal(n, a, x, ix, jx, descx, incx)
call pcscal(n, a, x, ix, jx, descx, incx)
call pzscal(n, a, x, ix, jx, descx, incx)
call pcsscal(n, a, x, ix, jx, descx, incx)
call pzdscal(n, a, x, ix, jx, descx, incx)
```

Include Files

- `mkl_pblas.h`

Description

The `p?scal` routines multiplies a n -element distributed vector `sub(x)` by the scalar `a`:

`sub(x) = a*sub(x)`,

where `sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx=m_x`, and `X(ix: ix+n-1, jx)` if `incx= 1`.

Input Parameters

`n`

(global) INTEGER. The length of distributed vector `sub(x)`, $n \geq 0$.

<i>a</i>	(global) REAL for psscal and pcsscal DOUBLE PRECISION for pdscal and pzdscal COMPLEX for pcscal DOUBLE COMPLEX for pzscal Specifies the scalar <i>a</i> .
<i>x</i>	(local) REAL for psscal DOUBLE PRECISION for pdscal COMPLEX for pcscal and pcsscal DOUBLE COMPLEX for pzscal and pzdscal Array, size $(j_x-1)*m_x + i_x+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten by the updated distributed vector <code>sub(x)</code>
----------	---

p?swap

Swaps two distributed vectors.

Syntax

```
call psswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

```
call pdswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

```
call pcswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

```
call pzswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

Given two distributed vectors `sub(x)` and `sub(y)`, the p?swap routines return vectors `sub(y)` and `sub(x)` swapped, each replacing the other.

Here `sub(x)` denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix:ix+n-1, jx)$ if $incx=1$;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy:iy+n-1, jy)$ if $incy=1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size $(j_x-1) * m_x + i_x + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size $(j_y-1) * m_y + i_y + (n-1) * \text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten by distributed vector $\text{sub}(y)$.
<i>y</i>	Overwritten by distributed vector $\text{sub}(x)$.

PBLAS Level 2 Routines

This section describes PBLAS Level 2 routines, which perform distributed matrix-vector operations. [Table "PBLAS Level 2 Routine Groups and Their Data Types"](#) lists the PBLAS Level 2 routine groups and the data types associated with them.

PBLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
<code>p?gemv</code>	s, d, c, z	Matrix-vector product using a distributed general matrix
<code>p?agemv</code>	s, d, c, z	Matrix-vector product using absolute values for a distributed general matrix
<code>p?ger</code>	s, d	Rank-1 update of a distributed general matrix
<code>p?gerc</code>	c, z	Rank-1 update (conjugated) of a distributed general matrix
<code>p?geru</code>	c, z	Rank-1 update (unconjugated) of a distributed general matrix
<code>p?hemv</code>	c, z	Matrix-vector product using a distributed Hermitian matrix
<code>p?ahemv</code>	c, z	Matrix-vector product using absolute values for a distributed Hermitian matrix
<code>p?her</code>	c, z	Rank-1 update of a distributed Hermitian matrix
<code>p?her2</code>	c, z	Rank-2 update of a distributed Hermitian matrix
<code>p?symv</code>	s, d	Matrix-vector product using a distributed symmetric matrix
<code>p?asymv</code>	s, d	Matrix-vector product using absolute values for a distributed symmetric matrix
<code>p?syr</code>	s, d	Rank-1 update of a distributed symmetric matrix
<code>p?syr2</code>	s, d	Rank-2 update of a distributed symmetric matrix
<code>p?trmv</code>	s, d, c, z	Distributed matrix-vector product using a triangular matrix
<code>p?atrmv</code>	s, d, c, z	Distributed matrix-vector product using absolute values for a triangular matrix
<code>p?trsv</code>	s, d, c, z	Solves a system of linear equations whose coefficients are in a distributed triangular matrix

p?gemv

Computes a distributed matrix-vector product using a general matrix.

Syntax

```
call psgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pdgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pcgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pzgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The `p?gemv` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y),$$

or

$$\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y),$$

or

$$\text{sub}(y) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(x) + \beta * \text{sub}(y),$$

where

alpha and *beta* are scalars,

sub(A) is a *m*-by-*n* submatrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are subvectors.

When *trans* = 'N' or 'n', *sub(x)* denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$

if *incx* = 1, *sub(y)* denotes $Y(\text{iy}, \text{jy}:\text{jy}+m-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+m-1, \text{jy})$ if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c', *sub(x)* denotes $X(\text{ix}, \text{jx}:\text{jx}+m-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+m-1, \text{jx})$ if *incx* = 1, *sub(y)* denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+m-1, \text{jy})$ if *incy* = 1.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y)$; if <i>trans</i> = 'T' or 't', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y)$; if <i>trans</i> = 'C' or 'c', then $\text{sub}(y) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(x) + \beta * \text{sub}(y)$.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(A)</i> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for <code>psgemv</code> DOUBLE PRECISION for <code>pdgemv</code> COMPLEX for <code>pcgemv</code> DOUBLE COMPLEX for <code>pzgemv</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for <code>psgemv</code> DOUBLE PRECISION for <code>pdgemv</code> COMPLEX for <code>pcgemv</code>

	DOUBLE COMPLEX for pzgemv
	Array, size $(lld_a, LOCq(ja+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	(local)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$ when <i>trans</i> = 'N' or 'n', and $(jx-1)*m_x + ix+(m-1)*abs(incx)$ otherwise. This array contains the entries of the distributed vector $sub(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $sub(y)$ need not be set on input.
<i>y</i>	(local)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Array, size $(jy-1)*m_y + iy+(m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(jy-1)*m_y + iy+(n-1)*abs(incy)$ otherwise. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.

<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector <i>sub(y)</i> .
----------	---

p?agemv

Computes a distributed matrix-vector product using absolute values for a general matrix.

Syntax

```
call psagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)
```

```
call pdagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)
```

```
call pcagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)
```

```
call pzagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The *p?agemv* routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)') * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

or

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)') * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

or

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{conjg}(\text{sub}(A)')) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where

alpha and *beta* are scalars,

sub(A) is a *m*-by-*n* submatrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are subvectors.

When *trans* = 'N' or 'n',

sub(x) denotes $X(\text{ix}:\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and

$X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx})$ if *incx* = 1,

sub(y) denotes $Y(\text{iy}:\text{iy}, \text{jy}:\text{jy}+m-1)$ if *incy* = *m_y*, and

$Y(\text{iy}:\text{iy}+m-1, \text{jy}:\text{jy})$ if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c',

$\text{sub}(x)$ denotes $X(ix:ix, jx:jx+m-1)$ if $incx = m_x$, and
 $X(ix:ix+m-1, jx:jx)$ if $incx = 1$,
 $\text{sub}(y)$ denotes $Y(iy:iy, jy:jy+n-1)$ if $incy = m_y$, and
 $Y(iy:iy+m-1, jy:jy)$ if $incy = 1$.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y) $ if <i>trans</i> = 'T' or 't', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y) $ if <i>trans</i> = 'C' or 'c', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y) $.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for psagemv DOUBLE PRECISION for pdagemv COMPLEX for pcagemv DOUBLE COMPLEX for pzagemv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for psagemv DOUBLE PRECISION for pdagemv COMPLEX for pcagemv DOUBLE COMPLEX for pzagemv Array, size $(lld_a, LOCq(ja+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local) REAL for psagemv DOUBLE PRECISION for pdagemv COMPLEX for pcagemv DOUBLE COMPLEX for pzagemv

Array, size $(j_x-1)*m_x + ix+(n-1)*abs(incx)$ when $trans = 'N'$ or $'n'$, and $(j_x-1)*m_x + ix+(m-1)*abs(incx)$ otherwise.

This array contains the entries of the distributed vector $sub(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.

descx

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

incx

(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

beta

(global) REAL for psagemv
 DOUBLE PRECISION for pdagemv
 COMPLEX for pcagemv
 DOUBLE COMPLEX for pzagemv

Specifies the scalar *beta*. When *beta* is set to zero, then $sub(y)$ need not be set on input.

y

(local) REAL for psagemv
 DOUBLE PRECISION for pdagemv
 COMPLEX for pcagemv
 DOUBLE COMPLEX for pzagemv

Array, size $(j_y-1)*m_y + iy+(m-1)*abs(incy)$ when $trans = 'N'$ or $'n'$, and $(j_y-1)*m_y + iy+(n-1)*abs(incy)$ otherwise.

This array contains the entries of the distributed vector $sub(y)$.

iy, jy

(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.

descy

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .

incy

(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

y

Overwritten by the updated distributed vector $sub(y)$.

p?ger

Performs a rank-1 update of a distributed general matrix.

Syntax

```
call psger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja,
desca)
```

```
call pdger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja,
desca)
```

Include Files

- mkl_pblas.h

Description

The `p?ger` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(y)^T + \text{sub}(A),$$

where:

alpha is a scalar,

sub(A) is a *m*-by-*n* distributed general matrix, $\text{sub}(A)=A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$,

sub(x) is an *m*-element distributed vector, *sub(y)* is an *n*-element distributed vector,

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+m-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+m-1, \text{jx})$ if *incx* = 1,

sub(y) denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if *incy* = 1.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(A)</i> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for <code>psger</code> DOUBLE REAL for <code>pdger</code> Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) REAL for <code>psger</code> DOUBLE REAL for <code>pdger</code> Array, size at least $(\text{jx}-1) * m_x + \text{ix} + (m-1) * \text{abs}(\text{incx})$. This array contains the entries of the distributed vector <i>sub(x)</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	(local) REAL for <code>psger</code>

	DOUBLE REAL for pdger Array, size at least $(jy-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local)REAL for psger DOUBLE REAL for pdger Array, size $(lld_a, LOCq(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	Overwritten by the updated distributed matrix $sub(A)$.
----------	--

p?gerc

Performs a rank-1 update (conjugated) of a distributed general matrix.

Syntax

```
call pcgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

```
call pzgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?gerc routines perform a distributed matrix-vector operation defined as

$$sub(A) := alpha * sub(x) * conjg(sub(y)') + sub(A),$$

where:

α is a scalar,

$\text{sub}(A)$ is a m -by- n distributed general matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$,

$\text{sub}(x)$ is an m -element distributed vector, $\text{sub}(y)$ is an n -element distributed vector,

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+m-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+m-1, \text{jx})$ if $\text{incx} = 1$,

$\text{sub}(y)$ denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if $\text{incy} = m_y$, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if $\text{incy} = 1$.

Input Parameters

m	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.
n	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
α	(global) COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Specifies the scalar α .
x	(local) COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Array, size at least $(\text{jx}-1) * m_x + \text{ix} + (n-1) * \text{abs}(\text{incx})$. This array contains the entries of the distributed vector $\text{sub}(x)$.
ix, jx	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
descx	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
incx	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . incx must not be zero.
y	(local) COMPLEX for pcgerc DOUBLE COMPLEX for pzgerc Array, size at least $(\text{jy}-1) * m_y + \text{iy} + (n-1) * \text{abs}(\text{incy})$. This array contains the entries of the distributed vector $\text{sub}(y)$.
iy, jy	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
descy	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
incy	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . incy must not be zero.
a	(local) COMPLEX for pcgerc

DOUBLE COMPLEX for pzgerc

Array, size at least $(lld_a, LOCq(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $sub(A)$.

ia, ja (global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

a Overwritten by the updated distributed matrix $sub(A)$.

p?geru

Performs a rank-1 update (unconjugated) of a distributed general matrix.

Syntax

```
call pcgeru(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

```
call pzgeru(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?geru routines perform a matrix-vector operation defined as

$$sub(A) := alpha * sub(x) * sub(y)^T + sub(A),$$

where:

alpha is a scalar,

$sub(A)$ is a m -by- n distributed general matrix, $sub(A) = A(ia:ia+m-1, ja:ja+n-1)$,

$sub(x)$ is an m -element distributed vector, $sub(y)$ is an n -element distributed vector,

$sub(x)$ denotes $X(ix, jx:jx+m-1)$ if $incx = m_x$, and $X(ix:ix+m-1, jx)$ if $incx = 1$,

$sub(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy:iy+n-1, jy)$ if $incy = 1$.

Input Parameters

m (global) INTEGER. Specifies the number of rows of the distributed matrix $sub(A)$, $m \geq 0$.

n (global) INTEGER. Specifies the number of columns of the distributed matrix $sub(A)$, $n \geq 0$.

alpha (global) COMPLEX for pcgeru

	DOUBLE COMPLEX for pzgeru Specifies the scalar <i>alpha</i> .
<i>x</i>	(local)COMPLEX for pcgeru DOUBLE COMPLEX for pzgeru Array, size at least $(j_x-1)*m_x + i_x+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $sub(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local)COMPLEX for pcgeru DOUBLE COMPLEX for pzgeru Array, size at least $(j_y-1)*m_y + i_y+(n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local)COMPLEX for pcgeru DOUBLE COMPLEX for pzgeru Array, size at least $(lld_a, LOCq(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	Overwritten by the updated distributed matrix $sub(A)$.
----------	--

p?hemv

Computes a distributed matrix-vector product using a Hermitian matrix.

Syntax

```
call pchemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
jy, descy, incy)
```

```
call pzhemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?hemv routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y),$$

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* Hermitian distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if *incx* = 1,

sub(y) denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if *incy* = 1.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size (<i>lld_a</i> , <i>LOCq(ja+n-1)</i>). This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry when <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <i>sub(A)</i>

is not referenced, and when $uplo = 'L'$ or $'l'$, the n -by- n lower triangular part of the distributed matrix $sub(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $sub(A)$ is not referenced.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size at least $(jx-1)*m_x + ix + (n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
<i>beta</i>	(global) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Specifies the scalar $beta$. When $beta$ is set to zero, then $sub(y)$ need not be set on input.
<i>y</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size at least $(jy-1)*m_y + iy + (n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector $sub(y)$.
----------	--

p?ahemv

Computes a distributed matrix-vector product using absolute values for a Hermitian matrix.

Syntax

```
call pcahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)
```

```
call pzahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)
```

Include Files

- mkl_pblas.h

Description

The p?ahemv routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* Hermitian distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if *incx* = 1,

sub(y) denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if *incy* = 1.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pcahemv DOUBLE COMPLEX for pzahemv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pcahemv DOUBLE COMPLEX for pzahemv Array, size (<i>lld_a</i> , LOCq(<i>ja</i> + <i>n</i> -1)). This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry when <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <i>sub(A)</i>

is not referenced, and when $uplo = 'L'$ or $'l'$, the n -by- n lower triangular part of the distributed matrix $sub(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $sub(A)$ is not referenced.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	(local) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Array, size at least $(jx-1)*m_x + ix + (n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
<i>beta</i>	(global) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Specifies the scalar $beta$. When $beta$ is set to zero, then $sub(y)$ need not be set on input.
<i>y</i>	(local) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Array, size at least $(jy-1)*m_y + iy + (n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector $sub(y)$.
----------	--

p?her

Performs a rank-1 update of a distributed Hermitian matrix.

Syntax

```
call pcher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

```
call pzher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?her routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{conjg}(\text{sub}(x)') + \text{sub}(A),$$

where:

alpha is a real scalar,

sub(A) is a *n*-by-*n* distributed Hermitian matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$,

sub(x) is distributed vector.

sub(x) denotes $X(ix, jx:jx+n-1)$ if *incx* = *m_x*, and $X(ix:ix+n-1, jx)$ if *incx* = 1.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for pcher DOUBLE REAL for pzher Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) COMPLEX for pcher DOUBLE COMPLEX for pzher Array, size at least $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector <i>sub(x)</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .

<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>a</i>	(local) COMPLEX for pcher DOUBLE COMPLEX for pzher Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry with <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated distributed matrix $\text{sub}(A)$. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.
----------	--

p?her2

Performs a rank-2 update of a distributed Hermitian matrix.

Syntax

```
call pcher2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

```
call pzher2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The p?her2 routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha \text{sub}(x) \text{conj}(\text{sub}(y)') + \text{conj}(\alpha) \text{sub}(y) \text{conj}(\text{sub}(x)') + \text{sub}(A),$$

where:

alpha is a scalar,

$\text{sub}(A)$ is a n -by- n distributed Hermitian matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

$\text{sub}(x)$ and $\text{sub}(y)$ are distributed vectors.

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$,

$\text{sub}(y)$ denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if $\text{incy} = m_y$, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if $\text{incy} = 1$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed Hermitian matrix $\text{sub}(A)$ is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the $\text{sub}(A)$ is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the $\text{sub}(A)$ is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pcher2 DOUBLE COMPLEX for pzher2 Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) COMPLEX for pcher2 DOUBLE COMPLEX for pzher2 Array, size at least $(\text{jx}-1) * m_x + \text{ix} + (n-1) * \text{abs}(\text{incx})$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for pcher2 DOUBLE COMPLEX for pzher2 Array, size at least $(\text{jy}-1) * m_y + \text{iy} + (n-1) * \text{abs}(\text{incy})$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .

<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) COMPLEX for <i>pcher2</i> DOUBLE COMPLEX for <i>pzher2</i> Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry with <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated distributed matrix $\text{sub}(A)$. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.
----------	--

p?symv

Computes a distributed matrix-vector product using a symmetric matrix.

Syntax

```
call pssymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pdsymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- `mkl_pblas.h`

Description

The `p?symv` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y),$$

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* symmetric distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$,

sub(y) denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if $\text{incy} = m_y$, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if $\text{incy} = 1$.

Input Parameters

<i>uplo</i>	<p>(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used:</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.</p>
<i>n</i>	<p>(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i>, $n \geq 0$.</p>
<i>alpha</i>	<p>(global) REAL for <i>pssymv</i> DOUBLE REAL for <i>pdsymv</i> Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>(local) REAL for <i>pssymv</i> DOUBLE REAL for <i>pdsymv</i> Array, size (<i>lld_a</i>, <i>LOCq(ja+n-1)</i>). This array contains the local pieces of the distributed matrix <i>sub(A)</i>.</p> <p>Before entry when <i>uplo</i> = 'U' or 'u', the <i>n</i>-by-<i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the <i>n</i>-by-<i>n</i> lower triangular part of the distributed matrix <i>sub(A)</i> must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of <i>sub(A)</i> is not referenced.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i>, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>x</i>	<p>(local) REAL for <i>pssymv</i> DOUBLE REAL for <i>pdsymv</i> Array, size at least $(\text{jx}-1) * m_x + \text{ix} + (n-1) * \text{abs}(\text{incx})$. This array contains the entries of the distributed vector <i>sub(x)</i>.</p>
<i>ix, jx</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i>, respectively.</p>

<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global) REAL for <code>pssymv</code> DOUBLE REAL for <code>pdsymv</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $\text{sub}(y)$ need not be set on input.
<i>y</i>	(local) REAL for <code>pssymv</code> DOUBLE REAL for <code>pdsymv</code> Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

y Overwritten by the updated distributed vector $\text{sub}(y)$.

p?asymv

Computes a distributed matrix-vector product using absolute values for a symmetric matrix.

Syntax

```
call psasymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

```
call pdasymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy, jy, descy, incy)
```

Include Files

- `mkl_pblas.h`

Description

The `p?symv` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where:

alpha and *beta* are scalars,

$\text{sub}(A)$ is a n -by- n symmetric distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

$\text{sub}(x)$ and $\text{sub}(y)$ are distributed vectors.

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$,

$\text{sub}(y)$ denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if $\text{incy} = m_y$, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if $\text{incy} = 1$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the $\text{sub}(A)$ is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the $\text{sub}(A)$ is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for psasymv DOUBLE REAL for pdasymv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for psasymv DOUBLE REAL for pdasymv Array, size (<i>lld_a</i> , $\text{LOCq}(\text{ja}+n-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry when <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and when <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	(local) REAL for psasymv DOUBLE PRECISION for pdasymv Array, size at least $(\text{jx}-1) * m_x + \text{ix} + (n-1) * \text{abs}(\text{incx})$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>beta</i>	(global)REAL for <i>psasymv</i> DOUBLE PRECISION for <i>pdasymv</i> Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>sub(y)</i> need not be set on input.
<i>y</i>	(local)REAL for <i>psasymv</i> DOUBLE PRECISION for <i>pdasymv</i> Array, size at least $(jy-1)*m_y + iy + (n-1)*abs(incy)$. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

y Overwritten by the updated distributed vector *sub(y)*.

p?syr

Performs a rank-1 update of a distributed symmetric matrix.

Syntax

```
call pssyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

```
call pdsyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

Include Files

- `mkl_pblas.h`

Description

The `p?syr` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(x)^T + \text{sub}(A),$$

where:

alpha is a scalar,

sub(A) is a *n*-by-*n* distributed symmetric matrix, *sub(A)* = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(x) is distributed vector.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx = m_x$, and $X(ix: ix+n-1, jx)$ if $incx = 1$,

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the $\text{sub}(A)$ is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the $\text{sub}(A)$ is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for pssyr DOUBLE REAL for pdsyr Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) REAL for pssyr DOUBLE REAL for pdsyr Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>a</i>	(local) REAL for pssyr DOUBLE REAL for pdsyr Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry with <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

- a* With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix *sub(A)*.
- With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix *sub(A)*.

p?syr2

Performs a rank-2 update of a distributed symmetric matrix.

Syntax

```
call pssyr2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

```
call pdsyr2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

Include Files

- mkl_pblas.h

Description

The *p?syr2* routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(y) + \alpha * \text{sub}(y) * \text{sub}(x) + \text{sub}(A),$$

where:

alpha is a scalar,

sub(A) is a *n*-by-*n* distributed symmetric matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if *incx* = 1,

sub(y) denotes $Y(\text{iy}, \text{jy}:\text{jy}+n-1)$ if *incy* = *m_y*, and $Y(\text{iy}:\text{iy}+n-1, \text{jy})$ if *incy* = 1.

Input Parameters

- uplo* (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed symmetric matrix *sub(A)* is used:
- If *uplo* = 'U' or 'u', then the upper triangular part of the *sub(A)* is used.
- If *uplo* = 'L' or 'l', then the low triangular part of the *sub(A)* is used.
- n* (global) INTEGER. Specifies the order of the distributed matrix *sub(A)*, $n \geq 0$.
- alpha* (global) REAL for *pssyr2*
DOUBLE REAL for *pdsyr2*

	Specifies the scalar <i>alpha</i> .
<i>x</i>	(local)REAL for pssyr2 DOUBLE REAL for pdsyr2 Array, size at least $(j_x-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <i>sub(x)</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	(local)REAL for pssyr2 DOUBLE REAL for pdsyr2 Array, size at least $(j_y-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.
<i>a</i>	(local)REAL for pssyr2 DOUBLE REAL for pdsyr2 Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry with <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the distributed symmetric matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and with <i>uplo</i> = 'L' or 'l', the <i>n</i> -by- <i>n</i> lower triangular part of the distributed matrix <i>sub(A)</i> must contain the lower triangular part of the distributed symmetric matrix and the strictly upper triangular part of <i>sub(A)</i> is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

a With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated distributed matrix *sub(A)*.

With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated distributed matrix *sub(A)*.

p?trmv

Computes a distributed matrix-vector product using a triangular matrix.

Syntax

```
call pstrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pdtrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The p?trmv routines perform one of the following distributed matrix-vector operations defined as

$\text{sub}(x) := \text{sub}(A) * \text{sub}(x)$, or $\text{sub}(x) := \text{sub}(A)' * \text{sub}(x)$, or $\text{sub}(x) := \text{conjg}(\text{sub}(A)') * \text{sub}(x)$,

where:

sub(A) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

sub(x) is an *n*-element distributed vector.

sub(x) denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if *incx* = *m_x*, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if *incx* = 1,

Input Parameters

uplo (global) CHARACTER*1. Specifies whether the distributed matrix *sub(A)* is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans (global) CHARACTER*1. Specifies the form of *op(sub(A))* used in the matrix equation:

if *transa* = 'N' or 'n', then $\text{sub}(x) := \text{sub}(A) * \text{sub}(x)$;

if *transa* = 'T' or 't', then $\text{sub}(x) := \text{sub}(A)' * \text{sub}(x)$;

if *transa* = 'C' or 'c', then $\text{sub}(x) := \text{conjg}(\text{sub}(A)') * \text{sub}(x)$.

<i>diag</i>	<p>(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular:</p> <p>if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular;</p> <p>if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.</p>
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>a</i>	<p>(local) REAL for pstrmv</p> <p>DOUBLE PRECISION for pdtrmv</p> <p>COMPLEX for pctrmv</p> <p>DOUBLE COMPLEX for pztrmv</p> <p>Array, size at least $(lld_a, LOCq(1, ja+n-1))$.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced .</p> <p>When <i>diag</i> = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	<p>(local) REAL for pstrmv</p> <p>DOUBLE PRECISION for pdtrmv</p> <p>COMPLEX for pctrmv</p> <p>DOUBLE COMPLEX for pztrmv</p> <p>Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$.</p> <p>This array contains the entries of the distributed vector $\text{sub}(x)$.</p>
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

x Overwritten by the transformed distributed vector `sub(x)`.

p?atrmv

Computes a distributed matrix-vector product using absolute values for a triangular matrix.

Syntax

```
call psatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)
```

```
call pdatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)
```

```
call pcatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)
```

```
call pzatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)
```

Include Files

- `mkl_pblas.h`

Description

The `p?atrmv` routines perform one of the following distributed matrix-vector operations defined as

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\text{beta} * \text{sub}(y))$, or

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)') * \text{abs}(\text{sub}(x)) + \text{abs}(\text{beta} * \text{sub}(y))$, or

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{conjg}(\text{sub}(A)')) * \text{abs}(\text{sub}(x)) + \text{abs}(\text{beta} * \text{sub}(y))$,

where:

α and beta are scalars,

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

$\text{sub}(x)$ is an n -element distributed vector.

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$.

Input Parameters

uplo (global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans (global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation:

if *trans* = 'N' or 'n', then $\text{sub}(y) := |\alpha| * |\text{sub}(A)| * |\text{sub}(x)| + |\text{beta} * \text{sub}(y)|$;

if $trans = 'T'$ or $'t'$, then $sub(y) := |alpha| * |sub(A)'| * |sub(x)| + |beta * sub(y)|$;
 if $trans = 'C'$ or $'c'$, then $sub(y) := |alpha| * |conjg(sub(A)')| * |sub(x)| + |beta * sub(y)|$.

diag (global) CHARACTER*1. Specifies whether the matrix $sub(A)$ is unit triangular:
 if $diag = 'U'$ or $'u'$ then the matrix is unit triangular;
 if $diag = 'N'$ or $'n'$, then the matrix is not unit triangular.

n (global) INTEGER. Specifies the order of the distributed matrix $sub(A)$, $n \geq 0$.

alpha (global) REAL for psatrmv
 DOUBLE PRECISION for pdatrmv
 COMPLEX for pcatrmv
 DOUBLE COMPLEX for pzatrmv
 Specifies the scalar *alpha*.

a (local) REAL for psatrmv
 DOUBLE PRECISION for pdatrmv
 COMPLEX for pcatrmv
 DOUBLE COMPLEX for pzatrmv
 Array, size at least $(lld_a, LOCq(1, ja+n-1))$.
 Before entry with $uplo = 'U'$ or $'u'$, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $sub(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $sub(A)$ is not referenced.
 Before entry with $uplo = 'L'$ or $'l'$, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $sub(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $sub(A)$ is not referenced.
 When $diag = 'U'$ or $'u'$, the local entries corresponding to the diagonal elements of the submatrix $sub(A)$ are not referenced either, but are assumed to be unity.

ia, ja (global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

x (local) REAL for psatrmv
 DOUBLE PRECISION for pdatrmv
 COMPLEX for pcatrmv
 DOUBLE COMPLEX for pzatrmv

	Array, size at least $(j_x-1)*m_x + ix+(n-1)*abs(incx)$.
	This array contains the entries of the distributed vector $sub(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global) REAL for psatrmv DOUBLE PRECISION for pdatrmv COMPLEX for pcatrmv DOUBLE COMPLEX for pzatrmv Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $sub(y)$ need not be set on input.
<i>y</i>	(local) REAL for psatrmv DOUBLE PRECISION for pdatrmv COMPLEX for pcatrmv DOUBLE COMPLEX for pzatrmv Array, size $(j_y-1)*m_y + iy+(m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(j_y-1)*m_y + iy+(n-1)*abs(incy)$ otherwise. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the transformed distributed vector $sub(y)$.
----------	--

p?trsv

Solves a system of linear equations whose coefficients are in a distributed triangular matrix.

Syntax

call pstrsv(*uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx*)

```
call pdtrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

Include Files

- mkl_pblas.h

Description

The `p?trsv` routines solve one of the systems of equations:

$\text{sub}(A) * \text{sub}(x) = b$, or $\text{sub}(A)' * \text{sub}(x) = b$, or $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$,

where:

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

b and $\text{sub}(x)$ are n -element distributed vectors,

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	(global) CHARACTER*1. Specifies the form of the system of equations: if <i>transa</i> = 'N' or 'n', then $\text{sub}(A) * \text{sub}(x) = b$; if <i>transa</i> = 'T' or 't', then $\text{sub}(A)' * \text{sub}(x) = b$; if <i>transa</i> = 'C' or 'c', then $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$.
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>a</i>	(local) REAL for <code>pstrsv</code> DOUBLE PRECISION for <code>pdtrsv</code> COMPLEX for <code>pctrsv</code> DOUBLE COMPLEX for <code>pztrsv</code> Array, size at least (<code>lld_a</code> , <code>LOCq(1, ja+n-1)</code>).

Before entry with $uplo = 'U'$ or $'u'$, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $sub(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $sub(A)$ is not referenced.

Before entry with $uplo = 'L'$ or $'l'$, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $sub(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $sub(A)$ is not referenced.

When $diag = 'U'$ or $'u'$, the local entries corresponding to the diagonal elements of the submatrix $sub(A)$ are not referenced either, but are assumed to be unity.

ia, ja	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
$desca$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
x	(local) REAL for $pstrsv$ DOUBLE PRECISION for $pdtrsv$ COMPLEX for $pctrsv$ DOUBLE COMPLEX for $pztrsv$ Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$. Before entry, $sub(x)$ must contain the n -element right-hand side distributed vector b .
ix, jx	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.
$descx$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

x	Overwritten with the solution vector.
-----	---------------------------------------

PBLAS Level 3 Routines

The PBLAS Level 3 routines perform distributed matrix-matrix operations. [Table "PBLAS Level 3 Routine Groups and Their Data Types"](#) lists the PBLAS Level 3 routine groups and the data types associated with them.

PBLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
<code>p?geadd</code>	s, d, c, z	Distributed matrix-matrix sum of general matrices
<code>p?tradd</code>	s, d, c, z	Distributed matrix-matrix sum of triangular matrices
<code>p?gemm</code>	s, d, c, z	Distributed matrix-matrix product of general matrices
<code>p?hemm</code>	c, z	Distributed matrix-matrix product, one matrix is Hermitian
<code>p?herk</code>	c, z	Rank-k update of a distributed Hermitian matrix
<code>p?her2k</code>	c, z	Rank-2k update of a distributed Hermitian matrix
<code>p?symm</code>	s, d, c, z	Matrix-matrix product of distributed symmetric matrices
<code>p?syrk</code>	s, d, c, z	Rank-k update of a distributed symmetric matrix
<code>p?syr2k</code>	s, d, c, z	Rank-2k update of a distributed symmetric matrix
<code>p?tran</code>	s, d	Transposition of a real distributed matrix
<code>p?tranc</code>	c, z	Transposition of a complex distributed matrix (conjugated)
<code>p?tranu</code>	c, z	Transposition of a complex distributed matrix
<code>p?trmm</code>	s, d, c, z	Distributed matrix-matrix product, one matrix is triangular
<code>p?trsm</code>	s, d, c, z	Solution of a distributed matrix equation, one matrix is triangular

p?geadd

Performs sum operation for two distributed general matrices.

Syntax

```
call psgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pcgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- `mkl_pblas.h`

Description

The `p?geadd` routines perform sum operation for two distributed general matrices. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{op}(\text{sub}(A)),$$

where:

`op(x)` is one of `op(x) = x`, or `op(x) = x'`,

`alpha` and `beta` are scalars,

`sub(C)` is an m -by- n distributed matrix, `sub(C) = C(ic:ic+m-1, jc:jc+n-1)`.

$\text{sub}(A)$ is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{op}(\text{sub}(A)) := \text{sub}(A)$; if <i>trans</i> = 'T' or 't', then $\text{op}(\text{sub}(A)) := \text{sub}(A)'$; if <i>trans</i> = 'C' or 'c', then $\text{op}(\text{sub}(A)) := \text{sub}(A)'$.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(C)$ and the number of columns of the submatrix $\text{sub}(A)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(C)$ and the number of rows of the submatrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Array, size $(\text{lld}_a, \text{LOCq}(\text{ja}+m-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd

Array, size $(lld_c, LOCq(jc+n-1))$.

This array contains the local pieces of the distributed matrix $sub(C)$.

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $sub(C)$, respectively.

desc

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c

Overwritten by the updated submatrix.

p?tradd

Performs sum operation for two distributed triangular matrices.

Syntax

```
call pstradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

```
call pdtradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

```
call pctradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

```
call pztradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

Include Files

- mkl_pblas.h

Description

The `p?tradd` routines perform sum operation for two distributed triangular matrices. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*op(sub(A)),
```

where:

$op(x)$ is one of $op(x) = x$, or $op(x) = x'$, or $op(x) = conjg(x')$.

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, $sub(C) = C(ic:ic+m-1, jc:jc+n-1)$.

sub(A) is a distributed matrix, $sub(A) = A(ia:ia+n-1, ja:ja+m-1)$.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the distributed matrix $sub(C)$ is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;

if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

(global) CHARACTER*1. Specifies the operation:

if *trans* = 'N' or 'n', then $op(sub(A)) := sub(A)$;

	if <i>trans</i> = 'T' or 't', then <code>op(sub(A)) := sub(A)'</code> ;
	if <i>trans</i> = 'C' or 'c', then <code>op(sub(A)) := conjg(sub(A)'</code>).
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <code>sub(C)</code> and the number of columns of the submatrix <code>sub(A)</code> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <code>sub(C)</code> and the number of rows of the submatrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global)REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local)REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Array, size (<code>lld_a</code> , <code>LOCq(ja+m-1)</code>). This array contains the local pieces of the distributed matrix <code>sub(A)</code> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global)REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <code>sub(C)</code> need not be set on input.
<i>c</i>	(local)REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Array, size (<code>lld_c</code> , <code>LOCq(jc+n-1)</code>). This array contains the local pieces of the distributed matrix <code>sub(C)</code> .

<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <i>sub(C)</i> , respectively.
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for distributed matrices.

Syntax

```
call psgemm(transa, transb, m, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pdgemm(transa, transb, m, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pcgemm(transa, transb, m, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pzgemm(transa, transb, m, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The `p?gemm` routines perform a matrix-matrix operation with general distributed matrices. The operation is defined as

$$\text{sub}(C) := \alpha * \text{op}(\text{sub}(A)) * \text{op}(\text{sub}(B)) + \beta * \text{sub}(C),$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$,

alpha and *beta* are scalars,

$\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+k-1)$, $\text{sub}(B) = B(\text{ib}:\text{ib}+k-1, \text{jb}:\text{jb}+n-1)$, and $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$, are distributed matrices.

Input Parameters

<i>transa</i>	(global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(\text{sub}(A)) = \text{sub}(A)$; if <i>transa</i> = 'T' or 't', then $\text{op}(\text{sub}(A)) = \text{sub}(A)'$; if <i>transa</i> = 'C' or 'c', then $\text{op}(\text{sub}(A)) = \text{sub}(A)'$.
---------------	---

<i>transb</i>	<p>(global) CHARACTER*1. Specifies the form of $op(sub(B))$ used in the matrix multiplication:</p> <p>if <i>transb</i> = 'N' or 'n', then $op(sub(B)) = sub(B)$;</p> <p>if <i>transb</i> = 'T' or 't', then $op(sub(B)) = sub(B)'$;</p> <p>if <i>transb</i> = 'C' or 'c', then $op(sub(B)) = sub(B)'$.</p>
<i>m</i>	<p>(global) INTEGER. Specifies the number of rows of the distributed matrices $op(sub(A))$ and $sub(C)$, $m \geq 0$.</p>
<i>n</i>	<p>(global) INTEGER. Specifies the number of columns of the distributed matrices $op(sub(B))$ and $sub(C)$, $n \geq 0$.</p> <p>The value of <i>n</i> must be at least zero.</p>
<i>k</i>	<p>(global) INTEGER. Specifies the number of columns of the distributed matrix $op(sub(A))$ and the number of rows of the distributed matrix $op(sub(B))$.</p> <p>The value of <i>k</i> must be greater than or equal to 0.</p>
<i>alpha</i>	<p>(global) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm</p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is equal to zero, then the local entries of the arrays <i>a</i> and <i>b</i> corresponding to the entries of the submatrices $sub(A)$ and $sub(B)$ respectively need not be set on input.</p>
<i>a</i>	<p>(local) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm</p> <p>Array, size <i>lld_a</i> by <i>kla</i>, where <i>kla</i> is $LOCc(ja+k-1)$ when <i>transa</i> = 'N' or 'n', and is $LOCq(ja+m-1)$ otherwise. Before entry this array must contain the local pieces of the distributed matrix $sub(A)$.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively</p>
<i>desca</i>	<p>(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>b</i>	<p>(local) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm</p>

	Array, size lld_b by $k1b$, where $k1b$ is $LOCc(jb+n-1)$ when $transb = 'N'$ or $'n'$, and is $LOCq(jb+k-1)$ otherwise. Before entry this array must contain the local pieces of the distributed matrix $sub(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $sub(B)$, respectively
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .
<i>beta</i>	(global)REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $sub(C)$ need not be set on input.
<i>c</i>	(local)REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Array, size $(lld_a, LOCq(jc+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $sub(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $sub(C)$, respectively
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

<i>c</i>	Overwritten by the m -by- n distributed matrix $alpha * op(sub(A)) * op(sub(B)) + beta * sub(C)$.
----------	---

p?hemm

Performs a scalar-matrix-matrix product (one matrix operand is Hermitian) and adds the result to a scalar-matrix product.

Syntax

```
call pchemm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
call pzhemm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The `p?hemm` routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

```
sub(C) := alpha*sub(A)*sub(B) + beta*sub(C),
```

or

```
sub(C) := alpha*sub(B)*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(A) is a Hermitian distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+m-1)$, if *side* = 'L', and $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

$\text{sub}(B) = B(\text{ib}:\text{ib}+m-1, \text{jb}:\text{jb}+n-1)$, $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER*1. Specifies whether the Hermitian distributed matrix <i>sub(A)</i> appears on the left or right in the operation: if <i>side</i> = 'L' or 'l', then $\text{sub}(C) := \text{alpha} * \text{sub}(A) * \text{sub}(B) + \text{beta} * \text{sub}(C)$; if <i>side</i> = 'R' or 'r', then $\text{sub}(C) := \text{alpha} * \text{sub}(B) * \text{sub}(A) + \text{beta} * \text{sub}(C)$.
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used; if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distribute submatrix <i>sub(C)</i> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distribute submatrix <i>sub(C)</i> , $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pchemm</code> DOUBLE COMPLEX for <code>pzhemm</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for <code>pchemm</code> DOUBLE COMPLEX for <code>pzhemm</code> Array, size (<code>lld_a</code> , <code>LOCq(ja+na-1)</code>). Before entry this array must contain the local pieces of the symmetric distributed matrix <i>sub(A)</i> , such that when <i>uplo</i> = 'U' or 'u', the <i>na</i> -by- <i>na</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly

lower triangular part of $\text{sub}(A)$ is not referenced, and when $\text{uplo} = 'L'$ or $'l'$, the na -by- na lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>b</i>	(local)COMPLEX for pchemm DOUBLE COMPLEX for pzhemm Array, size $(ll_d_b, LOCq(jb+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .
<i>beta</i>	(global)COMPLEX for pchemm DOUBLE COMPLEX for pzhemm Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local)COMPLEX for pchemm DOUBLE COMPLEX for pzhemm Array, size $(ll_d_c, LOCq(jc+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

<i>c</i>	Overwritten by the m -by- n updated distributed matrix.
----------	---

p?herk

Performs a rank- k update of a distributed Hermitian matrix.

Syntax

```
call pcherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?herk routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha*sub(A)*conjg(sub(A')) + beta*sub(C),
```

or

```
sub(C) := alpha*conjg(sub(A'))*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* Hermitian distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+n-1, \text{jc}:\text{jc}+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(A) = A(\text{ia}:\text{ia}+k-1, \text{ja}:\text{ja}+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{conjg}(\text{sub}(A')) + \beta * \text{sub}(C)$; if <i>trans</i> = 'C' or 'c', then $\text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A')) * \text{sub}(A) + \beta * \text{sub}(C)$.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	(global) INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrix <i>sub(A)</i> , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the distributed matrix <i>sub(A)</i> , $k \geq 0$.
<i>alpha</i>	(global) REAL for pcherk DOUBLE PRECISION for pzherk Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pcherk DOUBLE COMPLEX for pzherk

	Array, size (lld_a, kla) , where kla is $LOCq(ja+k-1)$ when $trans = 'N'$ or $'n'$, and is $LOCq(ja+n-1)$ otherwise. Before entry with $trans = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>beta</i>	(global) REAL for pcherk DOUBLE PRECISION for pzherk Specifies the scalar <i>beta</i> .
<i>c</i>	(local) COMPLEX for pcherk DOUBLE COMPLEX for pzherk Array, size $(lld_c, LOCq(jc+n-1))$. Before entry with $uplo = 'U'$ or $'u'$, this array contains n -by- n upper triangular part of the symmetric distributed matrix $sub(C)$ and its strictly lower triangular part is not referenced. Before entry with $uplo = 'L'$ or $'l'$, this array contains n -by- n lower triangular part of the symmetric distributed matrix $sub(C)$ and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $sub(C)$, respectively.
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

<i>c</i>	With $uplo = 'U'$ or $'u'$, the upper triangular part of $sub(C)$ is overwritten by the upper triangular part of the updated distributed matrix. With $uplo = 'L'$ or $'l'$, the lower triangular part of $sub(C)$ is overwritten by the upper triangular part of the updated distributed matrix.
----------	--

p?her2k

Performs a rank- $2k$ update of a Hermitian distributed matrix.

Syntax

```
call pcher2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pzher2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

Include Files

- `mkl_pblas.h`

Description

The `p?her2k` routines perform a distributed matrix-matrix operation defined as

$$\text{sub}(C) := \alpha * \text{sub}(A) * \text{conjg}(\text{sub}(B)') + \text{conjg}(\alpha) * \text{sub}(B) * \text{conjg}(\text{sub}(A)') + \beta * \text{sub}(C),$$

or

$$\text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(A) + \text{conjg}(\alpha) * \text{conjg}(\text{sub}(B)') * \text{sub}(B) + \beta * \text{sub}(C),$$

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* Hermitian distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+n-1, \text{jc}:\text{jc}+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(A) = A(\text{ia}:\text{ia}+k-1, \text{ja}:\text{ja}+n-1)$ otherwise.

sub(B) is a distributed matrix, $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(B) = B(\text{ib}:\text{ib}+k-1, \text{jb}:\text{jb}+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{conjg}(\text{sub}(B)') + \text{conjg}(\alpha) * \text{sub}(B) * \text{conjg}(\text{sub}(A)') + \beta * \text{sub}(C)$; if <i>trans</i> = 'C' or 'c', then $\text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(A) + \text{conjg}(\alpha) * \text{conjg}(\text{sub}(B)') * \text{sub}(B) + \beta * \text{sub}(C)$.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	(global) INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , and on entry with <i>trans</i> = 'C' or 'c', <i>k</i> specifies the number of rows of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , $k \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pcher2k</code> DOUBLE COMPLEX for <code>pzher2k</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for <code>pcher2k</code> DOUBLE COMPLEX for <code>pzher2k</code>

	Array, size (lld_a, kla) , where kla is $LOCq(ja+k-1)$ when $trans = 'N'$ or $'n'$, and is $LOCq(ja+n-1)$ otherwise. Before entry with $trans = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>b</i>	(local)COMPLEX for pcher2k DOUBLE COMPLEX for pzher2k Array, size (lld_b, klb) , where klb is $LOCq(jb+k-1)$ when $trans = 'N'$ or $'n'$, and is $LOCq(jb+n-1)$ otherwise. Before entry with $trans = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $sub(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $sub(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .
<i>beta</i>	(global)REAL for pcher2k DOUBLE PRECISION for pzher2k Specifies the scalar <i>beta</i> .
<i>c</i>	(local)COMPLEX for pcher2k DOUBLE COMPLEX for pzher2k Array, size $(lld_c, LOCq(jc+n-1))$. Before entry with $uplo = 'U'$ or $'u'$, this array contains n -by- n upper triangular part of the symmetric distributed matrix $sub(C)$ and its strictly lower triangular part is not referenced. Before entry with $uplo = 'L'$ or $'l'$, this array contains n -by- n lower triangular part of the symmetric distributed matrix $sub(C)$ and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $sub(C)$, respectively.
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

<i>c</i>	With $uplo = 'U'$ or $'u'$, the upper triangular part of $sub(C)$ is overwritten by the upper triangular part of the updated distributed matrix.
----------	---

With `uplo = 'L'` or `'l'`, the lower triangular part of `sub(C)` is overwritten by the upper triangular part of the updated distributed matrix.

p?symm

Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product for distributed matrices.

Syntax

```
call pssymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pdsymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pcsymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

```
call pzsymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descc)
```

Include Files

- `mkl_pblas.h`

Description

The `p?symm` routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

$$\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \beta \text{sub}(C),$$

or

$$\text{sub}(C) := \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C),$$

where:

alpha and *beta* are scalars,

sub(A) is a symmetric distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+m-1)$, if *side* = 'L', and $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

$\text{sub}(B) = B(\text{ib}:\text{ib}+m-1, \text{jb}:\text{jb}+n-1)$, $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$.

Input Parameters

- side* (global) CHARACTER*1. Specifies whether the symmetric distributed matrix *sub(A)* appears on the left or right in the operation:
- if *side* = 'L' or 'l', then $\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \beta \text{sub}(C)$;
- if *side* = 'R' or 'r', then $\text{sub}(C) := \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C)$.
- uplo* (global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix *sub(A)* is used:
- if *uplo* = 'U' or 'u', then the upper triangular part is used;

	if <code>uplo = 'L'</code> or <code>'l'</code> , then the lower triangular part is used.
<code>m</code>	(global) INTEGER. Specifies the number of rows of the distribute submatrix <code>sub(C)</code> , $m \geq 0$.
<code>n</code>	(global) INTEGER. Specifies the number of columns of the distribute submatrix <code>sub(C)</code> , $m \geq 0$.
<code>alpha</code>	(global)REAL for <code>pssymm</code> DOUBLE PRECISION for <code>pdsymm</code> COMPLEX for <code>pcsymm</code> DOUBLE COMPLEX for <code>pzsymm</code> Specifies the scalar <i>alpha</i> .
<code>a</code>	(local)REAL for <code>pssymm</code> DOUBLE PRECISION for <code>pdsymm</code> COMPLEX for <code>pcsymm</code> DOUBLE COMPLEX for <code>pzsymm</code> Array, size (<code>lld_a</code> , <code>LOCq(ja+na-1)</code>). Before entry this array must contain the local pieces of the symmetric distributed matrix <code>sub(A)</code> , such that when <code>uplo = 'U'</code> or <code>'u'</code> , the <i>na</i> -by- <i>na</i> upper triangular part of the distributed matrix <code>sub(A)</code> must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of <code>sub(A)</code> is not referenced, and when <code>uplo = 'L'</code> or <code>'l'</code> , the <i>na</i> -by- <i>na</i> lower triangular part of the distributed matrix <code>sub(A)</code> must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of <code>sub(A)</code> is not referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<code>b</code>	(local)REAL for <code>pssymm</code> DOUBLE PRECISION for <code>pdsymm</code> COMPLEX for <code>pcsymm</code> DOUBLE COMPLEX for <code>pzsymm</code> Array, size (<code>lld_b</code> , <code>LOCq(jb+n-1)</code>). Before entry this array must contain the local pieces of the distributed matrix <code>sub(B)</code> .
<code>ib, jb</code>	(global) INTEGER. The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix <code>sub(B)</code> , respectively.
<code>descb</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .

<i>beta</i>	(global)REAL for pssymm DOUBLE PRECISION for pdsymm COMPLEX for pcsymm DOUBLE COMPLEX for pzsymm Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <code>sub(C)</code> need not be set on input.
<i>c</i>	(local)REAL for pssymm DOUBLE PRECISION for pdsymm COMPLEX for pcsymm DOUBLE COMPLEX for pzsymm Array, size $(lld_c, LOCq(jc+n-1))$. Before entry this array must contain the local pieces of the distributed matrix <code>sub(C)</code> .
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.
<i>desc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.
----------	---

p?syrk

Performs a rank-*k* update of a symmetric distributed matrix.

Syntax

```
call pssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
call pdsyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
call pcsyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
call pzsyk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

Include Files

- mkl_pblas.h

Description

The `p?syrk` routines perform a distributed matrix-matrix operation defined as

$$\text{sub}(C) := \alpha * \text{sub}(A) * \text{sub}(A)^T + \beta * \text{sub}(C),$$

or

$$\text{sub}(C) := \alpha * \text{sub}(A)^T * \text{sub}(A) + \beta * \text{sub}(C),$$

where:

alpha and *beta* are scalars,

$\text{sub}(C)$ is an n -by- n symmetric distributed matrix, $\text{sub}(C)=C(ic:ic+n-1, jc:jc+n-1)$.

$\text{sub}(A)$ is a distributed matrix, $\text{sub}(A)=A(ia:ia+n-1, ja:ja+k-1)$, if $trans = 'N'$ or $'n'$, and $\text{sub}(A)=A(ia:ia+k-1, ja:ja+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	<p>(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ is used:</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the $\text{sub}(C)$ is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the $\text{sub}(C)$ is used.</p>
<i>trans</i>	<p>(global) CHARACTER*1. Specifies the operation:</p> <p>if <i>trans</i> = 'N' or 'n', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{sub}(A) + \beta * \text{sub}(C)$;</p> <p>if <i>trans</i> = 'T' or 't', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{sub}(A) + \beta * \text{sub}(C)$.</p>
<i>n</i>	<p>(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(C)$, $n \geq 0$.</p>
<i>k</i>	<p>(global) INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrix $\text{sub}(A)$, and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the distributed matrix $\text{sub}(A)$, $k \geq 0$.</p>
<i>alpha</i>	<p>(global) REAL for pssyrk DOUBLE PRECISION for pdsyrk COMPLEX for pcsyrk DOUBLE COMPLEX for pzsyk Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>(local) REAL for pssyrk DOUBLE PRECISION for pdsyrk COMPLEX for pcsyrk DOUBLE COMPLEX for pzsyk Array, size (<i>lld_a</i>, <i>kla</i>), where <i>kla</i> is $\text{LOCq}(ja+k-1)$ when <i>trans</i> = 'N' or 'n', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with <i>trans</i> = 'N' or 'n', this array contains the local pieces of the distributed matrix $\text{sub}(A)$.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>beta</i>	<p>(global) REAL for pssyrk</p>

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Specifies the scalar *beta*.

c (local) REAL for pssyrk

DOUBLE PRECISION for pdsyrk

COMPLEX for pcsyrk

DOUBLE COMPLEX for pzsyrk

Array, size (*lld_c*, LOCq(*jc+n-1*)).

Before entry with *uplo* = 'U' or 'u', this array contains *n*-by-*n* upper triangular part of the symmetric distributed matrix *sub(C)* and its strictly lower triangular part is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains *n*-by-*n* lower triangular part of the symmetric distributed matrix *sub(C)* and its strictly upper triangular part is not referenced.

ic, jc (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix *sub(C)*, respectively.

desc (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of *sub(C)* is overwritten by the upper triangular part of the updated distributed matrix.

With *uplo* = 'L' or 'l', the lower triangular part of *sub(C)* is overwritten by the upper triangular part of the updated distributed matrix.

p?syr2k

Performs a rank-2k update of a symmetric distributed matrix.

Syntax

```
call pssyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, desc)
```

```
call pdsyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, desc)
```

```
call pcsyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, desc)
```

```
call pzsyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, desc)
```

Include Files

- mkl_pblas.h

Description

The `p?syrr2k` routines perform a distributed matrix-matrix operation defined as

$$\text{sub}(C) := \alpha * \text{sub}(A) * \text{sub}(B)' + \alpha * \text{sub}(B) * \text{sub}(A)' + \beta * \text{sub}(C),$$

or

$$\text{sub}(C) := \alpha * \text{sub}(A)' * \text{sub}(B) + \alpha * \text{sub}(B)' * \text{sub}(A) + \beta * \text{sub}(C),$$

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* symmetric distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+n-1, \text{jc}:\text{jc}+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(A) = A(\text{ia}:\text{ia}+k-1, \text{ja}:\text{ja}+n-1)$ otherwise.

sub(B) is a distributed matrix, $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(B) = B(\text{ib}:\text{ib}+k-1, \text{jb}:\text{jb}+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{sub}(B)' + \alpha * \text{sub}(B) * \text{sub}(A)' + \beta * \text{sub}(C)$; if <i>trans</i> = 'T' or 't', then $\text{sub}(C) := \alpha * \text{sub}(B)' * \text{sub}(A) + \alpha * \text{sub}(A)' * \text{sub}(B) + \beta * \text{sub}(C)$.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	(global) INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , $k \geq 0$.
<i>alpha</i>	(global) REAL for <code>pssyr2k</code> DOUBLE PRECISION for <code>pdsyr2k</code> COMPLEX for <code>pcsyr2k</code> DOUBLE COMPLEX for <code>pzsyr2k</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for <code>pssyr2k</code> DOUBLE PRECISION for <code>pdsyr2k</code>

	COMPLEX for pcsyr2k
	DOUBLE COMPLEX for pzsyr2k
	Array, size (lld_a, kla) , where kla is $LOCq(ja+k-1)$ when $trans = 'N'$ or $'n'$, and is $LOCq(ja+n-1)$ otherwise. Before entry with $trans = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>b</i>	(local)REAL for pssyr2k DOUBLE PRECISION for pdsyr2k COMPLEX for pcsyr2k DOUBLE COMPLEX for pzsyr2k Array, size (lld_b, klb) , where klb is $LOCq(jb+k-1)$ when $trans = 'N'$ or $'n'$, and is $LOCq(jb+n-1)$ otherwise. Before entry with $trans = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $sub(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $sub(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .
<i>beta</i>	(global)REAL for pssyr2k DOUBLE PRECISION for pdsyr2k COMPLEX for pcsyr2k DOUBLE COMPLEX for pzsyr2k Specifies the scalar <i>beta</i> .
<i>c</i>	(local)REAL for pssyr2k DOUBLE PRECISION for pdsyr2k COMPLEX for pcsyr2k DOUBLE COMPLEX for pzsyr2k Array, size $(lld_c, LOCq(jc+n-1))$. Before entry with $uplo = 'U'$ or $'u'$, this array contains n -by- n upper triangular part of the symmetric distributed matrix $sub(C)$ and its strictly lower triangular part is not referenced. Before entry with $uplo = 'L'$ or $'l'$, this array contains n -by- n lower triangular part of the symmetric distributed matrix $sub(C)$ and its strictly upper triangular part is not referenced.

ic, jc (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix *sub(C)*, respectively.

descc (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of *sub(C)* is overwritten by the upper triangular part of the updated distributed matrix.
 With *uplo* = 'L' or 'l', the lower triangular part of *sub(C)* is overwritten by the upper triangular part of the updated distributed matrix.

p?tran

Transposes a real distributed matrix.

Syntax

```
call pstran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdtran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The *p?tran* routines transpose a real distributed matrix. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{sub}(A)'$$

where:

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$.

Input Parameters

m (global) INTEGER. Specifies the number of rows of the distributed matrix *sub(C)*, $m \geq 0$.

n (global) INTEGER. Specifies the number of columns of the distributed matrix *sub(C)*, $n \geq 0$.

alpha (global) REAL for *pstran*
 DOUBLE PRECISION for *pdtran*
 Specifies the scalar *alpha*.

a (local) REAL for *pstran*
 DOUBLE PRECISION for *pdtran*

	Array, size $(lld_a, LOCq(ja+m-1))$. This array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global)REAL for pstran DOUBLE PRECISION for pdtran Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $sub(C)$ need not be set on input.
<i>c</i>	(local)REAL for pstran DOUBLE PRECISION for pdtran Array, size $(lld_c, LOCq(jc+n-1))$. This array contains the local pieces of the distributed matrix $sub(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $sub(C)$, respectively.
<i>descc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tranu

Transposes a distributed complex matrix.

Syntax

```
call pctranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pztranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tranu routines transpose a complex distributed matrix. The operation is defined as

$$sub(C) := beta * sub(C) + alpha * sub(A)'$$

where:

alpha and *beta* are scalars,

$sub(C)$ is an *m*-by-*n* distributed matrix, $sub(C) = C(ic:ic+m-1, jc:jc+n-1)$.

$\text{sub}(A)$ is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(C)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(C)$, $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pctranu DOUBLE COMPLEX for pztranu Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pctranu DOUBLE COMPLEX for pztranu Array, size (<i>lld_a</i> , $\text{LOCq}(\text{ja}+m-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) COMPLEX for pctranu DOUBLE COMPLEX for pztranu Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local) COMPLEX for pctranu DOUBLE COMPLEX for pztranu Array, size (<i>lld_c</i> , $\text{LOCq}(\text{jc}+n-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.
<i>desc</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tranc

Transposes a complex distributed matrix, conjugated.

Syntax

```
call pctranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

```
call pztranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

Include Files

- mkl_pblas.h

Description

The p?tranc routines transpose a complex distributed matrix. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{conjg}(\text{sub}(A)'),$$

where:

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(C)</i> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pctranc DOUBLE COMPLEX for pztranc Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pctranc DOUBLE COMPLEX for pztranc Array, size (<i>lld_a</i> , LOCq(<i>ja</i> + <i>m</i> -1)). This array contains the local pieces of the distributed matrix <i>sub(A)</i> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) COMPLEX for pctranc DOUBLE COMPLEX for pztranc Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>sub(C)</i> need not be set on input.
<i>c</i>	(local) COMPLEX for pctranc DOUBLE COMPLEX for pztranc Array, size (<i>lld_c</i> , LOCq(<i>jc</i> + <i>n</i> -1)).

This array contains the local pieces of the distributed matrix $\text{sub}(C)$.

ic, jc (global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

desc (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c Overwritten by the updated submatrix.

p?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular) for distributed matrices.

Syntax

`call pstrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)`

`call pdtrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)`

`call pctrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)`

`call pztrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)`

Include Files

- `mkl_pblas.h`

Description

The `p?trmm` routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

`sub(B) := alpha*op(sub(A))*sub(B)`

or

`sub(B) := alpha*sub(B)*op(sub(A))`

where:

alpha is a scalar,

sub(B) is an m -by- n distributed matrix, $\text{sub}(B)=B(\text{ib}:\text{ib}+m-1, \text{jb}:\text{jb}+n-1)$.

A is a unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A)=A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+m-1)$, if *side* = 'L' or 'l', and $\text{sub}(A)=A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, if *side* = 'R' or 'r'.

op(sub(A)) is one of $\text{op}(\text{sub}(A)) = \text{sub}(A)$, or $\text{op}(\text{sub}(A)) = \text{sub}(A)'$, or $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.

Input Parameters

side (global) CHARACTER*1. Specifies whether $\text{op}(\text{sub}(A))$ appears on the left or right of $\text{sub}(B)$ in the operation:

if *side* = 'L' or 'l', then $\text{sub}(B) := \text{alpha}*\text{op}(\text{sub}(A))*\text{sub}(B)$;

	if <i>side</i> = 'R' or 'r', then $\text{sub}(B) := \alpha * \text{sub}(B) * \text{op}(\text{sub}(A))$.
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix <i>sub(A)</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	(global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then $\text{op}(\text{sub}(A)) = \text{sub}(A)$; if <i>transa</i> = 'T' or 't', then $\text{op}(\text{sub}(A)) = \text{sub}(A)'$; if <i>transa</i> = 'C' or 'c', then $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix <i>sub(A)</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(B)</i> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(B)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for <i>pstrmm</i> DOUBLE PRECISION for <i>pdtrmm</i> COMPLEX for <i>pctrmm</i> DOUBLE COMPLEX for <i>pztrmm</i> Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then the array <i>b</i> need not be set before entry.
<i>a</i>	(local) REAL for <i>pstrmm</i> DOUBLE PRECISION for <i>pdtrmm</i> COMPLEX for <i>pctrmm</i> DOUBLE COMPLEX for <i>pztrmm</i> Array, size <i>lld_a</i> by <i>ka</i> , where <i>ka</i> is at least $\text{LOCq}(1, ja+m-1)$ when <i>side</i> = 'L' or 'l' and is at least $\text{LOCq}(1, ja+n-1)$ when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix <i>sub(A)</i> , and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix <i>sub(A)</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix <i>sub(A)</i> , and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix <i>sub(A)</i> is not referenced.

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix *sub(A)* are not referenced either, but are assumed to be unity.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local)REAL for pstrmm DOUBLE PRECISION for pdtrmm COMPLEX for pctrmm DOUBLE COMPLEX for pztrmm Array, size (<i>lld_b</i> , LOCq(1, <i>jb+n-1</i>)). Before entry, this array contains the local pieces of the distributed matrix <i>sub(B)</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix <i>sub(B)</i> , respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	Overwritten by the transformed distributed matrix.
----------	--

p?trsm

Solves a distributed matrix equation (one matrix operand is triangular).

Syntax

```
call pstrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pdtrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pctrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pztrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
```

Include Files

- mkl_pblas.h

Description

The p?trsm routines solve one of the following distributed matrix equations:

$$\text{op}(\text{sub}(A)) * X = \text{alpha} * \text{sub}(B),$$

or

$$X * \text{op}(\text{sub}(A)) = \text{alpha} * \text{sub}(B),$$

where:

α is a scalar,

X and $\text{sub}(B)$ are m -by- n distributed matrices, $\text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1)$;

A is a unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1)$, if $side = 'L'$ or $'l'$, and $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, if $side = 'R'$ or $'r'$;

$\text{op}(\text{sub}(A))$ is one of $\text{op}(\text{sub}(A)) = \text{sub}(A)$, or $\text{op}(\text{sub}(A)) = \text{sub}(A)'$, or $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.

The distributed matrix $\text{sub}(B)$ is overwritten by the solution matrix X .

Input Parameters

<i>side</i>	(global) CHARACTER*1. Specifies whether $\text{op}(\text{sub}(A))$ appears on the left or right of X in the equation: if $side = 'L'$ or $'l'$, then $\text{op}(\text{sub}(A)) * X = \alpha * \text{sub}(B)$; if $side = 'R'$ or $'r'$, then $X * \text{op}(\text{sub}(A)) = \alpha * \text{sub}(B)$.
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular: if $uplo = 'U'$ or $'u'$, then the matrix is upper triangular; if $uplo = 'L'$ or $'l'$, then the matrix is low triangular.
<i>transa</i>	(global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation: if $transa = 'N'$ or $'n'$, then $\text{op}(\text{sub}(A)) = \text{sub}(A)$; if $transa = 'T'$ or $'t'$, then $\text{op}(\text{sub}(A)) = \text{sub}(A)'$; if $transa = 'C'$ or $'c'$, then $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if $diag = 'U'$ or $'u'$ then the matrix is unit triangular; if $diag = 'N'$ or $'n'$, then the matrix is not unit triangular.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(B)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(B)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for pstrsm DOUBLE PRECISION for pdtrsm COMPLEX for pctrsm DOUBLE COMPLEX for pztrsm Specifies the scalar α . When α is zero, then a is not referenced and b need not be set before entry.

<i>a</i>	<p>(local)REAL for pstrsm DOUBLE PRECISION for pdtrsm COMPLEX for pctrsm DOUBLE COMPLEX for pztrsm</p> <p>Array, size <i>lld_a</i> by <i>ka</i>, where <i>ka</i> is at least $\text{LOCq}(1, ja+m-1)$ when <i>side</i> = 'L' or 'l' and is at least $\text{LOCq}(1, ja+n-1)$ when <i>side</i> = 'R' or 'r'.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced .</p> <p>When <i>diag</i> = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.</p>
<i>desca</i>	<p>(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>b</i>	<p>(local)REAL for pstrsm DOUBLE PRECISION for pdtrsm COMPLEX for pctrsm DOUBLE COMPLEX for pztrsm</p> <p>Array, size (<i>lld_b</i>, $\text{LOCq}(1, jb+n-1)$).</p> <p>Before entry, this array contains the local pieces of the distributed matrix $\text{sub}(B)$.</p>
<i>ib, jb</i>	<p>(global) INTEGER. The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.</p>
<i>descb</i>	<p>(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i>.</p>

Output Parameters

<i>b</i>	Overwritten by the solution distributed matrix <i>X</i> .
----------	---

Partial Differential Equations Support

12

The Intel® Math Kernel Library (Intel® MKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see [Trigonometric Transform Routines](#)) and Poisson Solver (see [Fast Poisson Solver Routines](#)).

Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel MKL Fast Fourier Transform (FFT) interface (refer to [Fourier Transform Functions](#)), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the Intel MKL Poisson Solver. As it may be hard enough to modify the original code so as to make it work with Poisson Solver, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Solver routines can be called from C and Fortran.

NOTE

Intel MKL Trigonometric Transform and Poisson Solver routines support Fortran versions starting with Fortran 90.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in chapter "[Fast Fourier Transforms](#)", Intel® MKL supports the Real Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this manual, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. The current Intel MKL implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

To describe the Intel MKL TT interface, the C convention is used. Fortran users should refer to [Calling PDE Support Routines from Fortran](#).

For the list of Trigonometric Transforms currently implemented in Intel MKL TT interface, see [Transforms Implemented](#).

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel MKL wrappers without changing FFTW function calls in your code (refer to [FFTW to Intel® MKL Wrappers for FFTW 3.x](#) for details). However, you are strongly encouraged to use the native TT

interface for better performance. Another reason why you should use the wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{ki\pi}{n}, \quad k = 1, \dots, n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k) \sin \frac{ki\pi}{n}, \quad i = 1, \dots, n-1$$

Forward staggered sine transform

$$F(k) = \frac{1}{n} \sin \frac{(2k-1)\pi}{2} f(n) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{(2k-1)i\pi}{2n}, \quad k = 1, \dots, n$$

Backward staggered sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)i\pi}{2n}, \quad i = 1, \dots, n$$

Forward staggered2 sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad k = 1, \dots, n$$

Backward staggered2 sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad i = 1, \dots, n$$

Forward cosine transform

$$F(k) = \frac{1}{n} [f(0) + f(n) \cos k\pi] + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{ki\pi}{n}, \quad k = 0, \dots, n$$

Backward cosine transform

$$f(i) = \frac{1}{2} [F(0) + F(n) \cos i\pi] + \sum_{k=1}^{n-1} F(k) \cos \frac{ki\pi}{n}, \quad i = 0, \dots, n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{(2k+1)i\pi}{2n}, \quad k = 0, \dots, n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos \frac{(2k+1)i\pi}{2n}, \quad i = 0, \dots, n-1$$

Forward staggered2 cosine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \cos \frac{(2k-1)(2i-1)\pi}{4n}, \quad k = 1, \dots, n$$

Backward staggered2 cosine transform

$$f(i) = \sum_{k=1}^n F(k) \cos \frac{(2k-1)(2i-1)\pi}{4n}, \quad i = 1, \dots, n$$

NOTE

The size of the transform n can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. [Table "TT Interface Routines"](#) lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

TT Interface Routines

Routine	Description
<code>?_init_trig_transform</code>	Initializes basic data structures of Trigonometric Transforms.
<code>?_commit_trig_transform</code>	Checks consistency and correctness of user-defined data and creates a data structure to be used by Intel MKL FFT interface ¹ .
<code>?_forward_trig_transform</code> <code>?_backward_trig_transform</code>	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
<code>free_trig_transform</code>	Releases the memory used by a data structure needed for calling FFT interface ¹ .

¹TT routines call Intel MKL FFT interface for better performance.

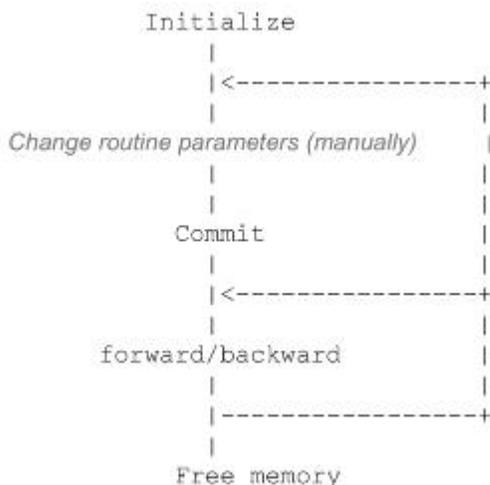
To find a transformed vector for a particular input vector only once, the Intel MKL TT interface routines are normally invoked in the order in which they are listed in [Table "TT Interface Routines"](#).

NOTE

Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking TT Interface Routines"](#) indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```

...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f. If you want to preserve the data stored in f,
save it to another location before the function call below */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...
  
```

You can find examples of code that uses TT interface routines to solve one-dimensional Helmholtz problem in the `examples\pdettf\source` folder in your Intel MKL directory.

Interface Description

All types in this documentation are either standard C types `float` and `double` or `MKL_INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and `INTEGER` or `INTEGER*8` integer type depending on the programming interface (LP64 or ILP64). To better understand usage of the types, see examples in the `examples\pdettf\source` folder in your Intel MKL directory.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays `ipar`, `dpar` and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL TT routines do not make copies of user input arrays.

NOTE

If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *Intel MKL User's Guide* (search the document for coding techniques to improve performance).

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel MKL FFT interface (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], double dpar[],
MKL_INT *stat);
```

```
void s_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], float spar[],
MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

<code>n</code>	MKL_INT*. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size $n+1$ for all but staggered2 transforms. Staggered2 transforms require the vector of size n .
<code>tt_type</code>	MKL_INT*. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: <code>MKL_SINE_TRANSFORM</code> , <code>MKL_STAGGERED_SINE_TRANSFORM</code> , <code>MKL_STAGGERED2_SINE_TRANSFORM</code> ; <code>MKL_COSINE_TRANSFORM</code> , <code>MKL_STAGGERED_COSINE_TRANSFORM</code> , <code>MKL_STAGGERED2_COSINE_TRANSFORM</code> .

Output Parameters

<code>ipar</code>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
-------------------	---

<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar[6]</i> . The status should be 0 to proceed to other TT routines.

Description

The `?_init_trig_transform` routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to `?_init_trig_transform`, all subsequently invoked TT routines use values of *ipar* and *dpar* (*spar*) array parameters returned by `?_init_trig_transform`. The routine initializes the entire array *ipar*. In the *dpar* or *spar* array, `?_init_trig_transform` initializes elements that do not depend upon the type of transform. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. You can skip a call to the initialization routine in your code. For more information, see [Caveat on Parameter Modifications](#).

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task.

?_commit_trig_transform

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_commit_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

<i>f</i>	double for <code>d_commit_trig_transform</code> , float for <code>s_commit_trig_transform</code> , array of size <i>n</i> for staggered2 transforms and of size <i>n</i> +1 for all other transforms, where <i>n</i> is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: <ul style="list-style-type: none"> • <i>f</i>[0] and <i>f</i>[<i>n</i>] for sine transforms • <i>f</i>[<i>n</i>] for staggered cosine transforms • <i>f</i>[0] for staggered sine transforms.
----------	---

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see [Fast Poisson Solver Routines](#)).

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

Output Parameters

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section " FFT Functions " in chapter "Fast Fourier Transforms").
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>dpar</i>	Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>spar</i>	Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: *handle*, *dpar* in case of `d_commit_trig_transform`, and *spar* in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of *dpar* or *spar* that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine performs only a basic check for correctness and consistency of the parameters. If you are going to modify parameters of TT routines, see the [Caveat on Parameter Modifications](#) section. Unlike `?_init_trig_transform`, you must call the `?_commit_trig_transform` routine in your code.

Return Values

<code>stat= 11</code>	The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning <code>ipar[6]=0</code> if you are sure that the parameters are correct.
<code>stat= 10</code>	The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning <code>ipar[6]=0</code> if you are sure that the parameters are correct.
<code>stat= 1</code>	The routine produced some warnings. You may proceed with computations by assigning <code>ipar[6]=0</code> if you are sure that the parameters are correct.
<code>stat= 0</code>	The routine completed the task normally.
<code>stat= -100</code>	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> • An error in the user's data was encountered. • Data in <code>ipar</code>, <code>dpar</code> or <code>spar</code> parameters became incorrect and/or inconsistent as a result of modifications.
<code>stat= -1000</code>	The routine stopped because of an FFT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.

NOTE

Although positive values of `stat` usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve `stat=0`.

?_forward_trig_transform

Computes the forward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_forward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_forward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

<i>f</i>	double for <code>d_forward_trig_transform</code> , float for <code>s_forward_trig_transform</code> , array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: <ul style="list-style-type: none"> • $f[0]$ and $f[n]$ for sine transforms • $f[n]$ for staggered cosine transforms • $f[0]$ for staggered sine transforms. Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).
<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, see FFT Functions).
<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the forward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_forward_trig_transform` with the *ipar* array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector *f* with the transformed vector.

NOTE

If you need a copy of the data vector f to be transformed, make the copy before calling the `?_forward_trig_transform` routine.

Return Values

<code>stat= 0</code>	The routine completed the task normally.
<code>stat= -100</code>	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> • An error in the user's data was encountered. • Data in <code>ipar</code>, <code>dpar</code> or <code>spar</code> parameters became incorrect and/or inconsistent as a result of modifications.
<code>stat= -1000</code>	The routine stopped because of an FFT interface error.
<code>stat= -10000</code>	The routine stopped because its commit step failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_backward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_backward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

f	double for <code>d_backward_trig_transform</code> , float for <code>s_backward_trig_transform</code> , array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors: <ul style="list-style-type: none"> • $f[0]$ and $f[n]$ for sine transforms • $f[n]$ for staggered cosine transforms • $f[0]$ for staggered sine transforms.
-----	---

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see [Fast Poisson Solver Routines](#)).

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, see FFT Functions).
<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the backward Trigonometric Transform of type defined in the [?_init_trig_transform](#) routine and passed to [?_backward_trig_transform](#) with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called [?_init_trig_transform](#) routine. The other data that facilitates the computation is created by [?_commit_trig_transform](#) and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine has a commit step, which calls the [?_commit_trig_transform](#) routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector *f* with the transformed vector.

NOTE

If you need a copy of the data vector *f* to be transformed, make the copy before calling the [?_backward_trig_transform](#) routine.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -100	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> An error in the user's data was encountered.

- Data in *ipar*, *dpar* or *spar* parameters became incorrect and/or inconsistent as a result of modifications.

stat = -1000

The routine stopped because of an FFT interface error.

stat = -10000

The routine stopped because its commit step failed to complete or the parameter *ipar*[0] was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by the FFT interface.

Syntax

```
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], MKL_INT *stat);
```

Include Files

- `mkl_trig_transforms.f90`

Input Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms").

Output Parameters

<i>handle</i>	The data structure used by Intel MKL FFT interface. Memory allocated for the structure is released on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The `free_trig_transform` routine cleans the memory used by the *handle* structure, needed for Intel MKL FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.

stat = -99999

The routine failed to complete the task.

Common Parameters

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.

NOTE

Initial values are assigned to the array parameters by the appropriate `?_init_trig_transform` and `?_commit_trig_transform` routines.

ipar MKL_INT array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in [Table "Elements of the ipar Array"](#):

Elements of the ipar Array

Index	Description
0	Contains the size of the problem to solve. The <code>?_init_trig_transform</code> routine sets <i>ipar</i> [0]= <i>n</i> , and all subsequently called TT routines use <i>ipar</i> [0] as the size of the transform.
1	Contains error messaging options: <ul style="list-style-type: none"> <i>ipar</i>[1]=-1 indicates that all error messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. <i>ipar</i>[1]=0 indicates that no error messages will be printed. <i>ipar</i>[1]=1 (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, each TT routine assigns a non-zero value to <i>stat</i> regardless of the <i>ipar</i> [1] setting.
2	Contains warning messaging options: <ul style="list-style-type: none"> <i>ipar</i>[2]=-1 indicates that all warning messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. <i>ipar</i>[2]=0 indicates that no warning messages will be printed. <i>ipar</i>[2]=1 (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen). In case of warnings, the <i>stat</i> parameter will acquire a non-zero value regardless of the <i>ipar</i> [2] setting.
3 through 4	Reserved for future use.
5	Contains the type of the transform. The <code>?_init_trig_transform</code> routine sets <i>ipar</i> [5]= <i>tt_type</i> , and all subsequently called TT routines use <i>ipar</i> [5] as the type of the transform.
6	Contains the <i>stat</i> value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with <i>stat</i> =0.

Index	Description
7	<p>Informs the <code>?_commit_trig_transform</code> routines whether to initialize data structures <code>dpar</code> (<code>spar</code>) and <code>handle</code>. <code>ipar[7]=0</code> indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1.</p> <p>The possibility to check correctness and consistency of input data without initializing data structures <code>dpar</code>, <code>spar</code> and <code>handle</code> enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that <code>ipar[7]</code> gives only if you are sure to have supplied proper tolerance value in the <code>dpar</code> or <code>spar</code> array. Otherwise, avoid tuning this parameter.</p>
8	<p>Contains message style options for TT routines. If <code>ipar[8]=0</code> then TT routines print all error and warning messages in Fortran-style notations. The default value is 1.</p> <p>When specifying message style options, be aware that by default, numbering of elements in Fortran arrays starts at 1. The use of <code>ipar[8]</code> enables you to view messages in a more convenient style.</p>
9	<p>Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Intel MKL Poisson Solver. The default value is 1. You are highly recommended not to alter this value. See also Caveat on Parameter Modifications.</p>
10	<p>Specifies the mode of compatibility with FFTW. The default value is 0. Set the value to 1 to invoke compatibility with FFTW. In the latter case, results will not be normalized, because FFTW does not do this. It is highly recommended not to alter this value, but rather use real-to-real FFTW to MKL wrappers, described in FFTW to Intel® MKL Wrappers for FFTW 3.x. See also Caveat on Parameter Modifications.</p>
11 through 127	Reserved for future use.

NOTE

While you can declare the `ipar` array as `MKL_INT ipar[11]`, for future compatibility you should declare `ipar` as `MKL_INT ipar[128]`.

Arrays `dpar` and `spar` are the same except in the data precision:

<code>dpar</code>	double array of size $5n/2+2$, holds data needed for double-precision routines to perform TT computations. This array is initialized in the <code>d_init_trig_transform</code> and <code>d_commit_trig_transform</code> routines.
<code>spar</code>	float array of size $5n/2+2$, holds data needed for single-precision routines to perform TT computations. This array is initialized in the <code>s_init_trig_transform</code> and <code>s_commit_trig_transform</code> routines.

As `dpar` and `spar` have similar elements in respective positions, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	<p>Contains the first absolute tolerance used by the appropriate <code>?_commit_trig_transform</code> routine. For a staggered cosine or a sine transform, $f[n]$ should be equal to 0.0 and for a staggered sine or a sine transform, $f[0]$ should be equal to 0.0. The <code>?_commit_trig_transform</code> routine checks whether absolute values of these parameters are below <code>dpar[0]*n</code> or <code>spar[0]*n</code>, depending on the routine precision. To suppress warnings resulting from tolerance checks, set <code>dpar[0]</code> or <code>spar[0]</code> to a sufficiently large number.</p>

Index	Description
1	Reserved for future use.
2 through $5n/2+1$	<p>Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform <i>tt_type</i>, set up in the <code>?_commit_trig_transform</code> routine:</p> <ul style="list-style-type: none"> • If <i>tt_type</i>=MKL_SINE_TRANSFORM, the transform uses only the first $n/2$ array elements, which contain tabulated sine values. • If <i>tt_type</i>=MKL_STAGGERED_SINE_TRANSFORM, the transform uses only the first $3n/2$ array elements, which contain tabulated sine and cosine values. • If <i>tt_type</i>=MKL_STAGGERED2_SINE_TRANSFORM, the transform uses all the $5n/2$ array elements, which contain tabulated sine and cosine values. • If <i>tt_type</i>=MKL_COSINE_TRANSFORM, the transform uses only the first n array elements, which contain tabulated cosine values. • If <i>tt_type</i>=MKL_STAGGERED_COSINE_TRANSFORM, the transform uses only the first $3n/2$ elements, which contain tabulated sine and cosine values. • If <i>tt_type</i>=MKL_STAGGERED2_COSINE_TRANSFORM, the transform uses all the $5n/2$ elements, which contain tabulated sine and cosine values.

NOTE

To save memory, you can define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of the TT interface enables you to skip a call to the `?_init_trig_transform` routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of *ipar*, *dpar* and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_trig_transform` routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the *ipar*, *spar* and *dpar* arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar* and *spar* arrays unless a strong need arises.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Optimization Notice

effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Implementation Details

Several aspects of the Intel MKL TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, Intel MKL provides you with the TT language-specific header file to include in your code:

- `mkl_trig_transforms.f90`, to be used together with `mkl_dfti.f90`.

NOTE

- Intel MKL TT interface supports Fortran versions starting with Fortran 90.
- Use of the Intel MKL TT software without including the above language-specific header files is not supported.

Header File

The header file below defines the following function prototypes:

```

SUBROUTINE D_INIT_TRIG_TRANSFORM(n, tt_type, ipar, dpar, stat)
  INTEGER, INTENT(IN) :: n, tt_type
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_INIT_TRIG_TRANSFORM

SUBROUTINE D_COMMIT_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_COMMIT_TRIG_TRANSFORM

SUBROUTINE D_FORWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_FORWARD_TRIG_TRANSFORM

SUBROUTINE D_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_BACKWARD_TRIG_TRANSFORM

```

```

SUBROUTINE S_INIT_TRIG_TRANSFORM(n, tt_type, ipar, spar, stat)
  INTEGER, INTENT(IN) :: n, tt_type
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_INIT_TRIG_TRANSFORM

SUBROUTINE S_COMMIT_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_COMMIT_TRIG_TRANSFORM

SUBROUTINE S_FORWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_FORWARD_TRIG_TRANSFORM

SUBROUTINE S_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_BACKWARD_TRIG_TRANSFORM

SUBROUTINE FREE_TRIG_TRANSFORM(handle, ipar, stat)
  INTEGER, INTENT(INOUT) :: ipar(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE FREE_TRIG_TRANSFORM

```

Fortran specifics of the TT routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran](#) section.

Fast Poisson Solver Routines

In addition to the Real Discrete Trigonometric Transforms (TT) interface (refer to [Trigonometric Transform Routines](#)), Intel MKL supports the the Poisson Solver interface. This interface implements a group of routines (Poisson Solver routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of a special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems that are solved by the Poisson Solver interface are defined more exactly in the [Poisson Solver Implementation](#) subsection. The Poisson Solver interface provides much flexibility of use: you can call routines with the default parameter values or adjust routines to your particular needs by manually tuning routine parameters. You can adjust the style of error and warning messages to a Fortran notation by setting up a dedicated parameter. This adds convenience to debugging, because you can read information in the way that is natural for your code. The Intel MKL Poisson Solver interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

To describe the Intel MKL Poisson Solver interface, the C convention is used. Fortran usage specifics can be found in the [Calling PDE Support Routines from Fortran](#) section.

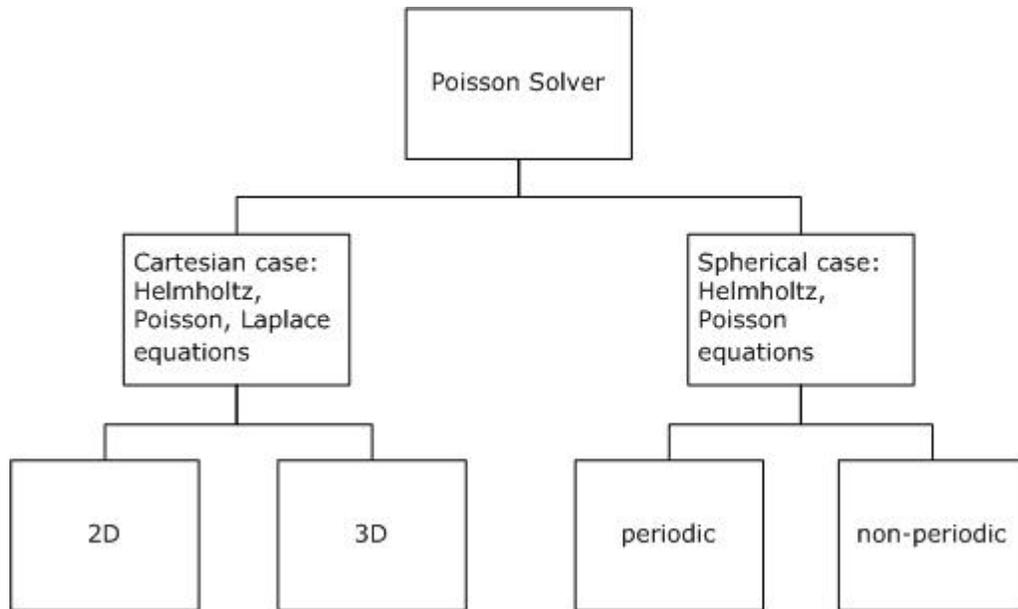
NOTE

Fortran users should keep in mind that array indices in Fortran start at 1 instead of 0, as they do in C.

Poisson Solver Implementation

Poisson Solver routines enable approximate solving of certain two-dimensional and three-dimensional problems. Figure "Structure of the Poisson Solver" shows the general structure of the Poisson Solver.

Structure of the Poisson Solver



NOTE

Although in the Cartesian case, both periodic and non-periodic solvers are also supported, they use the same interfaces.

Sections below provide details of the problems that can be solved using Intel MKL Poisson Solver.

Two-Dimensional Problems

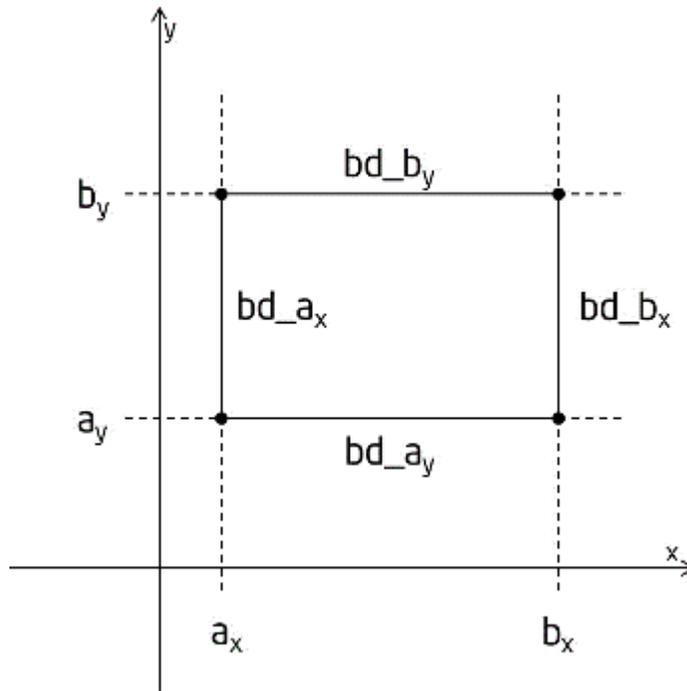
Notational Conventions

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x, a_y < y < b_y$ on a Cartesian plane:

$$bd_a_x = \{x = a_x, a_y \leq y \leq b_y\}, \quad bd_b_x = \{x = b_x, a_y \leq y \leq b_y\}$$

$$bd_a_y = \{a_x \leq x \leq b_x, y = a_y\}, \quad bd_b_y = \{a_x \leq x \leq b_x, y = b_y\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y , so bd_+ denotes any of the above boundaries.

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_\varphi < \varphi < b_\varphi, a_\theta < \theta < b_\theta$ on a sphere $0 \leq \varphi \leq 2\pi, 0 \leq \theta \leq \pi$:

$$bd_{a_\varphi} = \{\varphi = a_\varphi, a_\theta \leq \theta \leq b_\theta\}, \quad bd_{b_\varphi} = \{\varphi = b_\varphi, a_\theta \leq \theta \leq b_\theta\},$$

$$bd_{a_\theta} = \{a_\varphi \leq \varphi \leq b_\varphi, \theta = a_\theta\}, \quad bd_{b_\theta} = \{a_\varphi \leq \varphi \leq b_\varphi, \theta = b_\theta\}.$$

The wildcard "~" may stand for any of the symbols $a_\varphi, b_\varphi, a_\theta, b_\theta$, so bd_{\sim} denotes any of the above boundaries.

Two-dimensional Helmholtz problem on a Cartesian plane

The two-dimensional (2D) Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x, y), \quad q = const \geq 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x, a_y < y < b_y$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y) = G(x, y)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y) = g(x, y)$$

where

$$n = -x \text{ on } bd_{a_x}, \quad n = x \text{ on } bd_{b_x},$$

$$n = -y \text{ on } bd_{a_y}, \quad n = y \text{ on } bd_{b_y}.$$

- Periodic boundary conditions

$$u(a_x, y) = u(b_x, y), \quad \frac{\partial}{\partial x} u(a_x, y) = \frac{\partial}{\partial x} u(b_x, y),$$

$$u(x, a_y) = u(x, b_y), \frac{\partial}{\partial y} u(x, a_y) = \frac{\partial}{\partial y} u(x, b_y).$$

Two-dimensional Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 2D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

in a rectangle $a_x < x < b_x, a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ . In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y)=0$. The 2D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

in a rectangle $a_x < x < b_x, a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

$$-\Delta_s u + qu = f, \quad q = const \geq 0,$$

$$\Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \varphi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a domain bounded by angles $a_\varphi \leq \varphi \leq b_\varphi, a_\theta \leq \theta \leq b_\theta$ (spherical rectangle), with boundary conditions for particular domains listed in [Table "Details of Helmholtz Problem on a Sphere"](#).

Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\varphi - a_\varphi < 2 \pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary bd_+	<i>non-periodic</i>
Where $a_\varphi = 0, b_\varphi = 2 \pi,$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries bd_{a_θ} and bd_{b_θ}	<i>periodic</i>
Entire sphere, that is, $a_\varphi = 0, b_\varphi = 2 \pi, a_\theta = 0,$ and $b_\theta = \pi$	Boundary condition $\left(\sin \theta \frac{\partial u}{\partial \theta} \right) = 0$ $\theta \rightarrow 0$ $\theta \rightarrow \pi$ at the poles	<i>periodic</i>

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

$$-\Delta_s u = f, \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \varphi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a spherical rectangle $a_\varphi \leq \varphi \leq b_\varphi$, $a_\theta \leq \theta \leq b_\theta$ in cases listed in Table "Details of Helmholtz Problem on a Sphere". The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, in the rectangular domain a uniform mesh can be defined for the Cartesian case as:

$$\{x_i = a_x + ih_x, y_j = a_y + jh_y\},$$

$$i = 0, \dots, n_x, j = 0, \dots, n_y, h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}$$

and for the spherical case as:

$$\{\varphi_i = a_\varphi + ih_\varphi, \theta_j = a_\theta + jh_\theta\},$$

$$i = 0, \dots, n_\varphi, j = 0, \dots, n_\theta, h_\varphi = \frac{b_\varphi - a_\varphi}{n_\varphi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta}.$$

The Poisson Solver uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i, y_j) provided that you can supply the values of the right-hand side $f(x, y)$ in these points and the values of the appropriate boundary functions $G(x, y)$ and/or $g(x, y)$ in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points (φ_i, θ_j) provided that you can supply the values of the right-hand side $f(\varphi, \theta)$ in these points.

NOTE

The number of mesh intervals n_φ in the φ direction of a spherical mesh must be even in the periodic case. The Poisson Solver does not support spherical meshes that do not meet this condition.

Three-Dimensional Problems

Notational Conventions

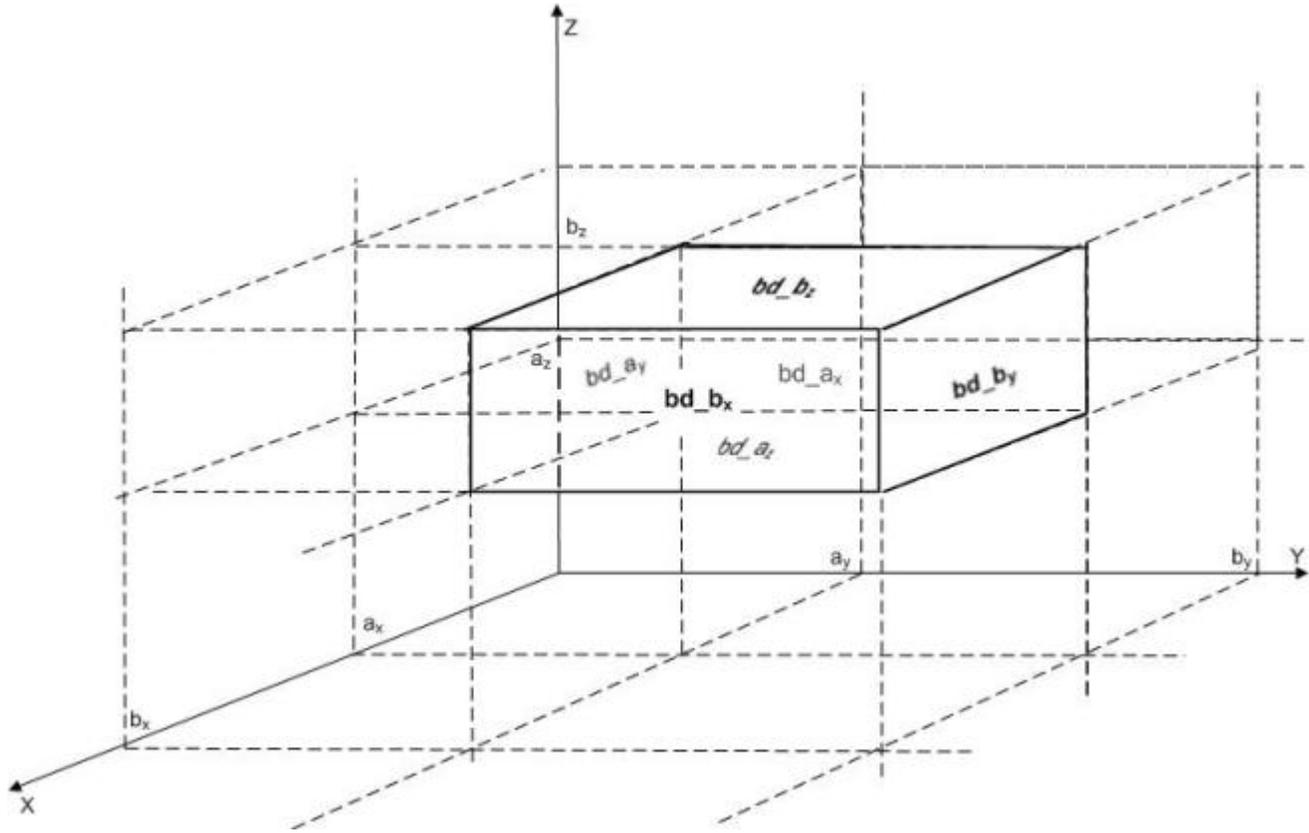
The Poisson Solver interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$:

$$bd_{a_x} = \{x = a_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}, bd_{b_x} = \{x = b_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\},$$

$$bd_{a_y} = \{a_x \leq x \leq b_x, y = a_y, a_z \leq z \leq b_z\}, bd_{b_y} = \{a_x \leq x \leq b_x, y = b_y, a_z \leq z \leq b_z\},$$

$$bd_{a_z} = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = a_z\}, bd_{b_z} = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = b_z\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y, a_z, b_z$, so bd_+ denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x, y, z), \quad q = const \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition
 $u(x, y, z) = G(x, y, z)$
- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y, z) = g(x, y, z)$$

where

$$n = -x \text{ on } bd_{a_x}, n = x \text{ on } bd_{b_x},$$

$$n = -y \text{ on } bd_{a_y}, n = y \text{ on } bd_{b_y},$$

$$n = -z \text{ on } bd_{a_z}, n = z \text{ on } bd_{b_z}.$$

- Periodic boundary conditions

$$u(a_x, y, z) = u(b_x, y, z), \quad \frac{\partial}{\partial x} u(a_x, y, z) = \frac{\partial}{\partial x} u(b_x, y, z),$$

$$u(x, a_y, z) = u(x, b_y, z), \quad \frac{\partial}{\partial y} u(x, a_y, z) = \frac{\partial}{\partial y} u(x, b_y, z),$$

$$u(x, y, a_z) = u(x, y, b_z), \quad \frac{\partial}{\partial z} u(x, y, a_z) = \frac{\partial}{\partial z} u(x, y, b_z).$$

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 3D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y, z)=0$. The 3D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh can be defined in the parallelepiped domain as:

$$\{x_i = a_x + ih_x, y_j = a_y + jh_y, z_k = a_z + kh_z\},$$

where

$$i = 0, \dots, n_x, j = 0, \dots, n_y, k = 0, \dots, n_z,$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}.$$

The Poisson Solver uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k) , provided that you can supply the values of the right-hand side $f(x, y, z)$ in these points and the values of the appropriate boundary functions $G(x, y, z)$ and/or $g(x, y, z)$ in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking Poisson Solver Routines

NOTE

This description always shows the solution process for the Helmholtz problem, because Fast Poisson Solvers and Fast Laplace Solvers are special cases of Fast Helmholtz Solvers (see [Poisson Solver Implementation](#)).

The Poisson Solver interface enables you to compute a solution of the Helmholtz problem in four steps. Each step is performed by a dedicated routine. [Table "Poisson Solver Interface Routines"](#) lists the routines and briefly describes their purpose.

Most Poisson Solver routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for the Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np".

Poisson Solver Interface Routines

Routine	Description
<code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code>	Initializes basic data structures for Fast Helmholtz Solver in the 2D/3D/periodic/non-periodic case, respectively.
<code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</code>	Checks consistency and correctness of input data and initializes data structures for the solver, including those used by the Intel MKL FFT interface ¹ .
<code>?_Helmholtz_2D/?_Helmholtz_3D/?_sph_p/?_sph_np</code>	Computes an approximate solution of the 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Solver Implementation) specified by the parameters.
<code>free_Helmholtz_2D/free_Helmholtz_3D/free_sph_p/free_sph_np</code>	Releases the memory used by the data structures needed for calling the Intel MKL FFT interface ¹ .

¹ Poisson Solver routines call the Intel MKL FFT interface for better performance.

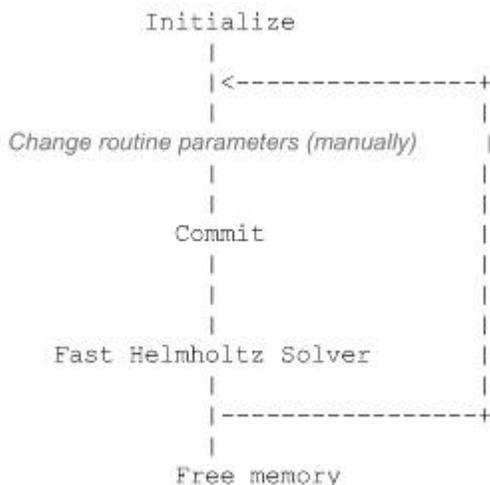
To find an approximate solution of Helmholtz problem only once, the Intel MKL Poisson Solver interface routines are normally invoked in the order in which they are listed in [Table "Poisson Solver Interface Routines"](#).

NOTE

Though the order of invoking Poisson Solver routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking Poisson Solver Routines"](#) indicates the typical order in which Poisson Solver routines can be invoked in a general case.

Typical Order of Invoking Poisson Solver Routines



A general scheme of using Poisson Solver routines for double-precision computations in a 3D Cartesian case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below from the 3D to 2D case by changing the ending of the Poisson Solver routine names.

```

...
d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, Bctype, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */

```

```

/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar,
&stat);
d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar, &stat);
free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...

```

A general scheme of using Poisson Solver routines for double-precision computations in a spherical periodic case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below to a scheme for a non-periodic case by changing the ending of the Poisson Solver routine names from "p" to "np".

```

...
d_init_sph_p(&ap, &bp, &at, &bt, &np, &nt, &q, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
d_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
free_sph_p(&handle_s, &handle_c, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...

```

You can find examples of code that uses Poisson Solver routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the `examples\pdepoissonf\source` folder in your Intel MKL directory.

Interface Description

All numerical types in this section are either standard C types `float` and `double` or MKL `INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and either `INTEGER` or `INTEGER*8` integer type depending on the programming interface (LP64 or ILP64). To better understand usage of the types, see examples in the `examples\pdepoissonf\source` folder in your Intel MKL directory.

Routine Options

All Poisson Solver routines use parameters for passing various options to the routines. These parameters are arrays `ipar`, `dpar`, and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs. For more details, see the descriptions of specific routines.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

Poisson Solver routines take arrays of user data as input. For example, the `d_Helmholtz_3D` routine takes user arrays to compute an approximate solution to the 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL Poisson Solver routines do not make copies of user input arrays.

NOTE

If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *Intel MKL User's Guide* (search the document for coding techniques to improve performance).

Routines for the Cartesian Solver

The section describes Poisson Solver routines for the Cartesian case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and 2D and 3D.

NOTE

Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

Poisson Solver routines call Intel MKL FFT routines (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhance performance of the Poisson Solver routines.

?_init_Helmholtz_2D/?_init_Helmholtz_3D

Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

```
void d_init_Helmholtz_2D (const double * ax, const double * bx, const double * ay,
const double * by, const MKL_INT * nx, const MKL_INT * ny, const char * Bctype, const
double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat);
```

```
void s_init_Helmholtz_2D (const float * ax, const float * bx, const float * ay, const
float * by, const MKL_INT * nx, const MKL_INT * ny, const char * Bctype, const float *
q, MKL_INT * ipar, float * spar, MKL_INT * stat);
```

```
void d_init_Helmholtz_3D (const double * ax, const double * bx, const double * ay,
const double * by, const double * az, const double * bz, const MKL_INT * nx, const
MKL_INT * ny, const MKL_INT * nz, const char * Bctype, const double * q, MKL_INT
*ipar, double * dpar, MKL_INT * stat);
```

```
void s_init_Helmholtz_3D (const float * ax, const float * bx, const float * ay, const
float * by, const float * az, const float * bz, const MKL_INT * nx, const MKL_INT *
ny, const MKL_INT * nz, const char * Bctype, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat);
```

Include Files

- mkl_poisson.f90

Input Parameters

ax double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
The coordinate of the leftmost boundary of the domain along the x-axis.

bx double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
The coordinate of the rightmost boundary of the domain along the x-axis.

ay double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,

	float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the leftmost boundary of the domain along the y-axis.
<i>by</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the rightmost boundary of the domain along the y-axis.
<i>az</i>	double* for d_init_Helmholtz_3D, float* for s_init_Helmholtz_3D.
	The coordinate of the leftmost boundary of the domain along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>bz</i>	double* for d_init_Helmholtz_3D, float* for s_init_Helmholtz_3D.
	The coordinate of the rightmost boundary of the domain along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>nx</i>	MKL_INT*. The number of mesh intervals along the x-axis.
<i>ny</i>	MKL_INT*. The number of mesh intervals along the y-axis.
<i>nz</i>	MKL_INT*. The number of mesh intervals along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>BCtype</i>	char*. Contains the type of boundary conditions on each boundary. Must contain four characters for ?_init_Helmholtz_2D and six characters for ?_init_Helmholtz_3D. Each of the characters can be 'N' (Neumann boundary condition), 'D' (Dirichlet boundary condition), or 'P' (periodic boundary conditions). Specify the types of boundary conditions for the boundaries in the following order: <i>bd_ax</i> , <i>bd_bx</i> , <i>bd_ay</i> , <i>bd_by</i> , <i>bd_az</i> , and <i>bd_bz</i> . Specify periodic boundary conditions on the respective boundaries in pairs (for example, 'PPDD' or 'NNPP' in the 2D case). The types of boundary conditions for the last two boundaries are needed only in the 3D case.
<i>q</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of <i>q</i> to 0.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).
<i>dpar</i>	double array of size $5 \cdot nx/2 + 7$ in the 2D case or $5 \cdot (nx + ny)/2 + 9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).

<i>spar</i>	float array of size $5 \cdot n_x / 2 + 7$ in the 2D case or $5 \cdot (n_x + n_y) / 2 + 9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routines initialize basic data structures for Poisson Solver computations of the appropriate precision. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in [Common Parameters](#).

CAUTION

Data structures initialized and created by 2D flavors of the routine cannot be used by 3D flavors of any Poisson Solver routines, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

[_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#)

Checks consistency and correctness of input data and initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx,
const double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT *
ipar, double * dpar, MKL_INT * stat );

void s_commit_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const
float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar,
float * spar, MKL_INT * stat );

void d_commit_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx,
const double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
double * dpar, MKL_INT * stat );

void s_commit_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const
float * bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
float * spar, MKL_INT * stat );
```

Include Files

- `mk1_poisson.f90`

Input Parameters

<code>f</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code>.</p> <p>Contains the right-hand side of the problem packed in a single vector:</p> <ul style="list-style-type: none"> • 2D problem: The size of the vector for the is $(n_x+1)*(n_y+1)$. The value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(n_x+1)]$. • 3D problem: The size of the vector for the is $(n_x+1)*(n_y+1)*(n_z+1)$. The value of the right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(n_x+1)+k*(n_x+1)*(n_y+1)]$. <p>Note that to solve the Laplace problem, you should set all the elements of the array <code>f</code> to 0.</p> <p>Note also that the array <code>f</code> may be altered by the routine. To preserve the <code>f</code> vector, save it to another memory location.</p>
<code>ipar</code>	<p>MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver (for details, refer to <code>ipar</code>).</p>
<code>dpar</code>	<p>double array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*n_x/2+7$ • 3D problem: $5*(n_x+n_y)/2+9$ <p>Contains double-precision data to be used by the Fast Helmholtz Solver (for details, refer to <code>dpar</code> and <code>spar</code>).</p>
<code>spar</code>	<p>float array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*n_x/2+7$ • 3D problem: $5*(n_x+n_y)/2+9$ <p>Contains single-precision data to be used by the Fast Helmholtz Solver (for details, refer to <code>dpar</code> and <code>spar</code>).</p>
<code>bd_ax</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code>.</p> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis (for more information, refer to a detailed description of <code>bd_ax</code>).</p>
<code>bd_bx</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code>.</p> <p>Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis (for more information, refer to a detailed description of <code>bd_bx</code>).</p>
<code>bd_ay</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code>.</p>

Contains values of the boundary condition on the leftmost boundary of the domain along the *y*-axis (for more information, refer to [a detailed description of *bd_ay*](#)).

bd_by double* for *d_commit_Helmholtz_2D/d_commit_Helmholtz_3D*,
float* for *s_commit_Helmholtz_2D/s_commit_Helmholtz_3D*.

Contains values of the boundary condition on the rightmost boundary of the domain along the *y*-axis (for more information, refer to [a detailed description of *bd_by*](#)).

bd_az double* for *d_commit_Helmholtz_3D*,
float* for *s_commit_Helmholtz_3D*.

Used only by *?_commit_Helmholtz_3D*. Contains values of the boundary condition on the leftmost boundary of the domain along the *z*-axis (for more information, refer to [a detailed description of *bd_az*](#)).

bd_bz double* for *d_commit_Helmholtz_3D*,
float* for *s_commit_Helmholtz_3D*.

Used only by *?_commit_Helmholtz_3D*. Contains values of the boundary condition on the rightmost boundary of the domain along the *z*-axis (for more information, refer to [a detailed description of *bd_bz*](#)).

Output Parameters

f Contains right-hand side of the problem, possibly altered on output.

ipar Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in [ipar](#).

dpar Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in [dpar](#) and [spar](#).

spar Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in [dpar](#) and [spar](#).

xhandle, yhandle DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"). *yhandle* is used only by *?_commit_Helmholtz_3D*.

stat MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The *?_commit_Helmholtz_2D/?_commit_Helmholtz_3D* routines check the consistency and correctness of the parameters to be passed to the solver routines *?_Helmholtz_2D/?_Helmholtz_3D*. They also initialize the *xhandle* and *yhandle* data structures, *ipar* array, and *dpar* or *spar* array, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the *?_commit_Helmholtz_2D/?_commit_Helmholtz_3D* routines initialize and to what values these elements are initialized.

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine, at this point results in an incorrect solution.

- 2D problem: the size of the vector is $(nx+1)*(ny+1)$. The value of the modified right-hand side in the mesh point (i, j) is stored in $r[i+j*(nx+1)]$.
- 3D problem: the size of the vector is $(nx+1)*(ny+1)*(nz+1)$. The value of the modified right-hand side in the mesh point (i, j, k) is stored in $r[i+j*(nx+1)+k*(nx+1)*(ny+1)]$.

xhandle, yhandle

DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). *yhandle* is used only by ?_Helmholtz_3D.

ipar

MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to *ipar*).

dpar

double array of size depending on the dimension of the problem:

- 2D problem: $5*nx/2+7$
- 3D problem: $5*(nx+ny)/2+9$

Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to *dpar* and *spar*).

spar

float array of size depending on the dimension of the problem:

- 2D problem: $5*nx/2+7$
- 3D problem: $5*(nx+ny)/2+9$

Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to *dpar* and *spar*).

bd_ax

double* for d_Helmholtz_2D/d_Helmholtz_3D,

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis (for more information, refer to a [detailed description of *bd_ax*](#)).

bd_bx

double* for d_Helmholtz_2D/d_Helmholtz_3D,

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis (for more information, refer to a [detailed description of *bd_bx*](#)).

bd_ay

double* for d_Helmholtz_2D/d_Helmholtz_3D,

float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the y -axis for more information, refer to [a detailed description of `bd_ay`](#)).

`bd_by` double* for `d_Helmholtz_2D/d_Helmholtz_3D`,
float* for `s_Helmholtz_2D/s_Helmholtz_3D`.

Contains values of the boundary condition on the rightmost boundary of the domain along the y -axis (for more information, refer to [a detailed description of `bd_by`](#)).

`bd_az` double* for `d_Helmholtz_3D`,
float* for `s_Helmholtz_3D`.

Used only by `?_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along the z -axis (for more information, refer to [a detailed description of `bd_az`](#)).

`bd_bz` double* for `d_Helmholtz_3D`,
float* for `s_Helmholtz_3D`.

Used only by `?_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along the z -axis (for more information, refer to [a detailed description of `bd_bz`](#)).

NOTE

To avoid incorrect computation results, do not change arrays `bd_ax`, `bd_bx`, `bd_ay`, `bd_by`, `bd_az`, `bd_bz` between a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine and a subsequent call to the appropriate `?_Helmholtz_2D/?_Helmholtz_3D` routine.

Output Parameters

`f` On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

`xhandle, yhandle` Data structures used by the Intel MKL FFT interface. Although the addresses do not change, the structures are modified on output.

`ipar` Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in [ipar](#).

`stat` MKL_INT*. Routine completion status, which is also written to `ipar[0]`. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_Helmholtz_2D/?_Helmholtz_3D` routines compute the approximate solution of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the [Poisson Solver Implementation](#) section. The `f` parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of `ax`, `bx`, `ay`, `by`, `az`, and `bz` are passed to the routines with the `spar/dpar` array, and values of `nx`, `ny`, `nz`, and `BCTYPE` are passed with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with some warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -2</code>	The routine stopped because division by zero occurred. It usually happens if the data in the <code>dpar</code> or <code>spar</code> array was altered by mistake.
<code>stat= -3</code>	The routine stopped because the sufficient memory was unavailable for the computations.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of the Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, MKL_INT* stat);
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE*
yhandle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- `mkl_poisson.f90`

Input Parameters

<code>xhandle, yhandle</code>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structure <code>yhandle</code> is used only by <code>free_Helmholtz_3D</code> .
<code>ipar</code>	MKL_INT array of size 128. Contains integer data used by Fast Helmholtz Solver (for details, refer to <code>ipar</code>).

Output Parameters

<i>xhandle, yhandle</i>	Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.
<i>ipar</i>	Contains integer data used by Fast Helmholtz Solver. On output, the status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The `free_Helmholtz_2D-free_Helmholtz_3D` routine releases the memory used by the *xhandle* and *yhandle* structures, which are needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -1000	The routine stopped because of an Intel MKL FFT or TT interface error.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

Routines for the Spherical Solver

The section describes Poisson Solver routines for the spherical case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and periodic (having names ending in "p") and non-periodic (having names ending in "np").

These Poisson Solver routines also call the Intel MKL FFT routines (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhance the performance of the Poisson Solver routines.

?_init_sph_p/?_init_sph_np

Initializes basic data structures of the periodic and non-periodic Fast Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p (const double * ap, const double * at, const double * bp, const
double * bt, const MKL_INT * np, const MKL_INT *nt, const double * q, MKL_INT * ipar,
double * dpar, MKL_INT * stat );
```

```
void s_init_sph_p (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

```
void d_init_sph_np (const double * ap, const double * at, const double * bp, const
double * bt, const MKL_INT * np, const MKL_INT *nt, const double * q, MKL_INT * ipar,
double * dpar, MKL_INT * stat );
```

```
void s_init_sph_np (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

Include Files

- `mkl_poisson.f90`

Input Parameters

<i>ap</i>	<p>double* for <code>d_init_sph_p/d_init_sph_np</code>, float* for <code>s_init_sph_p/s_init_sph_np</code>.</p> <p>The coordinate (angle) of the leftmost boundary of the domain along the φ-axis.</p>
<i>bp</i>	<p>double* for <code>d_init_sph_p/d_init_sph_np</code>, float* for <code>s_init_sph_p/s_init_sph_np</code>.</p> <p>The coordinate (angle) of the rightmost boundary of the domain along the φ-axis.</p>
<i>at</i>	<p>double* for <code>d_init_sph_p/d_init_sph_np</code>, float* for <code>s_init_sph_p/s_init_sph_np</code>.</p> <p>The coordinate (angle) of the leftmost boundary of the domain along the θ-axis.</p>
<i>bt</i>	<p>double* for <code>d_init_sph_p/d_init_sph_np</code>, float* for <code>s_init_sph_p/s_init_sph_np</code>.</p> <p>The coordinate (angle) of the rightmost boundary of the domain along the θ-axis.</p>
<i>np</i>	MKL_INT*. The number of mesh intervals along the φ -axis. Must be even in the periodic case.
<i>nt</i>	MKL_INT*. The number of mesh intervals along the θ -axis.
<i>q</i>	<p>double* for <code>d_init_sph_p/d_init_sph_np</code>, float* for <code>s_init_sph_p/s_init_sph_np</code>.</p> <p>The constant Helmholtz coefficient. To solve the Poisson problem, set the value of <i>q</i> to 0.</p>

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <code>ipar[0]</code> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_sph_p/?_init_sph_np` routines initialize basic data structures for Poisson Solver computations. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the `ipar`, `dpar`, and `spar` array parameters returned by the routine. A detailed description of the array parameters can be found in [Common Parameters](#).

CAUTION

Data structures initialized and created by periodic flavors of the routine cannot be used by non-periodic flavors of any Poisson Solver routines for Helmholtz Solver on a sphere, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

`stat= 0`

The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this `stat` value.

`stat= -99999`

The routine failed to complete the task because of fatal error.

`?_commit_sph_p/?_commit_sph_np`

Checks consistency and correctness of input data and initializes certain data structures required to solve the periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s,
DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_commit_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_commit_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double*
dpar, MKL_INT* stat);

void s_commit_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float*
spar, MKL_INT* stat);
```

Include Files

- `mkl_poisson.f90`

Input Parameters

`f`

double* for `d_commit_sph_p/d_commit_sph_np`,
float* for `s_commit_sph_p/s_commit_sph_np`.

Contains the right-hand side of the problem packed in a single vector. The size of the vector is $(np+1)*(nt+1)$ and value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(np+1)]$.

Note that the array `f` may be altered by the routine. Save this vector to another memory location if you want to preserve it.

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to ipar).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

<i>f</i>	Contains the right-hand side of the problem, possibly altered on output.
<i>ipar</i>	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar .
<i>dpar</i>	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>spar</i>	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>handle_s, handle_c, handle</i>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section " FFT Functions " in chapter "Fast Fourier Transforms"). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_commit_sph_p</code> and <i>handle</i> is used only in <code>?_commit_sph_np</code> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar</i> [0]. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_commit_sph_p/?_commit_sph_np` routines check consistency and correctness of the parameters to be passed to the solver routines `?_sph_p/?_sph_np`, respectively. They also initialize certain data structures. The routine `?_commit_sph_p` initializes structures *handle_s* and *handle_c*, and `?_commit_sph_np` initializes *handle*. The routines also initialize the *ipar* array and *dpar* or *spar* array, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_sph_p/?_commit_sph_np` routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see [Caveat on Parameter Modifications](#). Unlike `?_init_sph_p/?_init_sph_np`, you must call the `?_commit_sph_p/?_commit_sph_np` routines. Values of *np* and *nt* are passed to each of the routines with the *ipar* array.

Return Values

<i>stat</i> = 1	The routine completed without errors but with warnings.
<i>stat</i> = 0	The routine successfully completed the task.

<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

?_sph_p/?_sph_np

Computes the solution of the spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar,
MKL_INT* stat);

void s_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar,
MKL_INT* stat);
```

Include Files

- `mkl_poisson.f90`

Input Parameters

<code>f</code>	double* for <code>d_sph_p/d_sph_np</code> , float* for <code>s_sph_p/s_sph_np</code> . Contains the right-hand side of the problem packed in a single vector and modified by the appropriate <code>?_commit_sph_p/?_commit_sph_np</code> routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the <code>?_commit_sph_p/?_commit_sph_np</code> routine, at this point results in an incorrect solution. The size of the vector is $(np+1)*(nt+1)$ and the value of the modified right-hand side in the mesh point (i, j) is stored in <code>f[i+j*(np+1)]</code> .
<code>handle_s, handle_c, handle</code>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). <code>handle_s</code> and <code>handle_c</code> are used only in <code>?_sph_p</code> and <code>handle</code> is used only in <code>?_sph_np</code> .
<code>ipar</code>	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to <code>ipar</code>).

<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

<i>f</i>	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
<i>handle_s</i> , <i>handle_c</i> , <i>handle</i>	Data structures used by the Intel MKL FFT interface.
<i>ipar</i>	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar .
<i>dpar</i>	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>spar</i>	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar</i> [0]. Continue to call other Poisson Solver routines only if the status is 0.

Description

The *sph_p/sph_np* routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to the formulas given in the [Poisson Solver Implementation](#) section. The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *np* and *nt* are passed to each of the routines with the *ipar* array.

Return Values

<i>stat</i> = 1	The routine completed without errors but with warnings.
<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -2	The routine stopped because division by zero occurred. It usually happens if the data in the <i>dpar</i> or <i>spar</i> array was altered by mistake.
<i>stat</i> = -3	The routine stopped because the memory was insufficient to complete the computations.
<i>stat</i> = -100	The routine stopped because an error in the input data was found or the data in the <i>dpar</i> , <i>spar</i> , or <i>ipar</i> array was altered by mistake.
<i>stat</i> = -1000	The routine stopped because of an Intel MKL FFT or TT interface error.

`stat= -10000`

The routine stopped because the initialization failed to complete or the parameter `ipar[0]` was altered by mistake.

`stat= -99999`

The routine failed to complete the task because of a fatal error.

free_sph_p/free_sph_np

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_sph_p(DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c,
MKL_INT* ipar, MKL_INT* stat);
```

```
void free_sph_np(DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- `mkl_poisson.f90`

Input Parameters

`handle_s`, `handle_c`,
`handle`

DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structures `handle_s` and `handle_c` are used only in `free_sph_p`, and `handle` is used only in `free_sph_np`.

`ipar`

MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to [ipar](#)).

Output Parameters

`handle_s`, `handle_c`,
`handle`

Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.

`ipar`

Contains integer data to be used by Fast Helmholtz Solver on a sphere. On output, the status of the routine call is written to `ipar[0]`.

`stat`

MKL_INT*. Routine completion status, which is also written to `ipar[0]`.

Description

The `free_sph_p/free_sph_np` routine releases the memory used by the `handle_s`, `handle_c` or `handle` structures, needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

`stat= 0`

The routine successfully completed the task.

`stat= -1000`

The routine stopped because of an Intel MKL FFT or TT interface error.

stat= -99999

The routine failed to complete the task because of a fatal error.

Common Parameters

ipar

ipar MKL_INT array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in [Table "Elements of the ipar Array"](#):

NOTE

Initial values are assigned to the array parameters by the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` and `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np` routines.

Elements of the ipar Array

Index	Description
0	Contains status value of the last Poisson Solver routine called. In general, it should be 0 on exit from a routine to proceed with the Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</code> routines of how the Commit step of the computation should be carried out (see Figure "Typical Order of Invoking Poisson Solver Routines"). A non-zero value of <code>ipar[0]</code> with decimal representation

$$\overline{abc} = 100a + 10b + c$$

=100a+10b+c, where each of *a*, *b*, and *c* is equal to 0 or 9, indicates that some parts of the Commit step should be omitted.

- If *c*=9, the routine omits checking of parameters and initialization of the data structures.
- If *b*=9,
 - In the Cartesian case, the routine omits the adjustment of the right-hand side vector *f* to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function *g*) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function *G*).
 - For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the *dpar/spar* array.
- If *a*=9, the routine omits the normalization of the right-hand side vector *f*. Depending on the solver, the normalization means:
 - 2D Cartesian case: multiplication by h_y^2 , where h_y is the mesh size in the *y* direction (for details, see [Poisson Solver Implementation](#)).
 - 3D (Cartesian) case: multiplication by h_z^2 , where h_z is the mesh size in the *z* direction.
 - Helmholtz solver on a sphere: multiplication by h_θ^2 , where h_θ is the mesh size in the θ direction (for details, see [Poisson Solver Implementation](#)).

Using `ipar[0]` you can adjust the routine to your needs and improve efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious when using this method, because any misunderstanding of the commit process may cause incorrect results or program failure (see also [Caveat on Parameter Modifications](#)).

Index	Description
1	<p>Contains error messaging options:</p> <ul style="list-style-type: none"> • $ipar[1]=-1$ indicates that all error messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device (usually, screen). • $ipar[1]=0$ indicates that no error messages will be printed. • $ipar[1]=1$ is the default value. It indicates that all error messages are printed to the standard output device. <p>In case of errors, the <code>stat</code> parameter contains a non-zero value on exit from a routine regardless of the $ipar[1]$ setting.</p>
2	<p>Contains warning messaging options:</p> <ul style="list-style-type: none"> • $ipar[2]=-1$ indicates that all warning messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • $ipar[2]=0$ indicates that no warning messages will be printed. • $ipar[2]=1$ is the default value. It indicates that all warning messages are printed to the standard output device. <p>In case of warnings, the <code>stat</code> parameter contains a non-zero value on exit from a routine regardless of the $ipar[2]$ setting.</p>
3	Unused.
Parameters 4 through 9 are used only in the Cartesian case.	
4	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if $BCtype[0]='P'$ • 1, if $BCtype[0]='N'$ • 0, if $BCtype[0]='D'$ • -1, otherwise
5	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if $BCtype[1]='P'$ • 1, if $BCtype[1]='N'$ • 0, if $BCtype[1]='D'$ • -1, otherwise
6	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if $BCtype[2]='P'$ • 1, if $BCtype[2]='N'$ • 0, if $BCtype[2]='D'$ • -1, otherwise
7	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if $BCtype[3]='P'$ • 1, if $BCtype[3]='N'$ • 0, if $BCtype[3]='D'$ • -1, otherwise
8	Takes this value:

Index	Description
	<ul style="list-style-type: none"> • 2, if $BCType[4]='P'$ • 1, if $BCType[4]='N'$ • 0, if $BCType[4]='D'$ • -1, otherwise
9	Takes this value: <ul style="list-style-type: none"> • 2, if $BCType[5]='P'$ • 1, if $BCType[5]='N'$ • 0, if $BCType[5]='D'$ • -1, otherwise
10	Takes the value of <ul style="list-style-type: none"> • n_x, that is, the number of intervals along the x-axis, in the Cartesian case. • n_p, that is, the number of intervals along the φ-axis, in the spherical case.
11	Takes the value of <ul style="list-style-type: none"> • n_y, that is, the number of intervals along the y-axis, in the Cartesian case • n_t, that is, the number of intervals along the θ-axis, in the spherical case.
12	Takes the value of n_z , the number of intervals along the z -axis. This parameter is used only in the 3D case (Cartesian).
13	Has the value of 6, which specifies the internal partitioning of the $dpar/spar$ array.
14	Takes the value of $ipar[13]+ipar[10]+1$, which specifies the internal partitioning of the $dpar/spar$ array.

The values of $ipar[15] - ipar[20]$ depend on the dimension of the problem for the Cartesian solver or on whether the solver on a sphere is periodic.

	Cartesian Solver		Spherical Solver	
	2D case	3D case	Periodic case	Non-periodic case
15	Unused	Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.		
16	Unused	Takes the value of $ipar[14]+ipar[11]+1$, which specifies the internal partitioning of the $dpar/spar$ array.		
17	Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+1$, which specifies the internal partitioning of the $dpar/spar$ array.		
18	Takes the value of $ipar[14]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.

Index	Description			
19	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.		Unused
20	Takes the value of $ipar[19]+3*ipar[12]/4$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[18]+3*ipar[11]/2+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Takes the value of $ipar[18]+3*ipar[10]/4+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array.	Unused

The values of $ipar[21] - ipar[119]$ are assigned regardless of the dimension of the problem for the Cartesian solver or of whether the solver on a sphere is periodic.

21	Contains message style options: <ul style="list-style-type: none"> $ipar[21]=0$ indicates that Poisson Solver routines print all error and warning messages in Fortran-style notations. 			
22	Contains the number of OpenMP threads to be used for computations in a multithreaded environment. The default value is 1 in the serial mode, and the result returned by the <code>mkl_get_max_threads</code> function otherwise.			
23	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic Cartesian case.			
24	Takes the value of $ipar[23]+3*ipar[12]/4$, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic Cartesian case.			
25	Takes the value of $ipar[20]+1$, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic 3D Cartesian case.			
26	Takes the value of $ipar[25]+3*ipar[13]/4$, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic 3D Cartesian case.			
27 through 39	Unused.			
40 through 59	Contain the first twenty elements of the <i>ipar</i> array of the first Trigonometric Transform that the solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)			
60 through 79	Contain the first twenty elements of the <i>ipar</i> array of the second Trigonometric Transform that the 3D Cartesian and periodic spherical solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)			
80 through 99	Contain the first twenty elements of the <i>ipar</i> array of the third Trigonometric Transform that the solver uses in case of periodic boundary conditions along the <i>x</i> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)			
100 through 119	Contain the first twenty elements of the <i>ipar</i> array of the fourth Trigonometric Transform used by periodic spherical solvers and 3D Cartesian solvers with periodic boundary conditions along the <i>y</i> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)			

NOTE

While you can declare the `ipar` array as `MKL_INT ipar[120]`, for future compatibility you should declare `ipar` as `MKL_INT ipar[128]`.

dpar and spar

Arrays `dpar` and `spar` are the same except in the data precision:

`dpar`

Holds data needed for double-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `double` array of size $5 \cdot n_x / 2 + 7$ in the 2D case or $5 \cdot (n_x + n_y) / 2 + 9$ in the 3D case; initialized in the `d_init_Helmholtz_2D/`
`d_init_Helmholtz_3D` and `d_commit_Helmholtz_2D/`
`d_commit_Helmholtz_3D` routines.
- For the spherical solver, `double` array of size $5 \cdot n_p / 2 + n_t + 10$; initialized in the `d_init_sph_p/d_init_sph_np` and `d_commit_sph_p/`
`d_commit_sph_np` routines.

`spar`

Holds data needed for single-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, `float` array of size $5 \cdot n_x / 2 + 7$ in the 2D case or $5 \cdot (n_x + n_y) / 2 + 9$ in the 3D case; initialized in the `s_init_Helmholtz_2D/`
`s_init_Helmholtz_3D` and `s_commit_Helmholtz_2D/`
`s_commit_Helmholtz_3D` routines.
- For the spherical solver, `float` array of size $5 \cdot n_p / 2 + n_t + 10$; initialized in the `s_init_sph_p/s_init_sph_np` and `s_commit_sph_p/s_commit_sph_np` routines.

Because `dpar` and `spar` have similar elements in each position, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	<p>In the Cartesian case, contains the length of the interval along the x-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_x in the x direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the φ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_φ in the φ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
1	<p>In the Cartesian case, contains the length of the interval along the y-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_y in the y direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the θ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_θ in the θ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
2	<p>In the Cartesian case, contains the length of the interval along the z-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_z in the z direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the Cartesian solver, this parameter is used only in the 3D case.</p>

Index	Description
3	<p>In the spherical solver, contains the coordinate of the leftmost boundary along the θ-axis after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine.</p>
3	<p>Contains the value of the coefficient q after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.</p>
4	<p>Contains the tolerance parameter after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.</p> <ul style="list-style-type: none"> <li data-bbox="420 470 1458 730">• In the Cartesian case, this value is used only for the pure Neumann boundary conditions (<code>BCtype="NNNN"</code> in the 2D case; <code>BCtype="NNNNNN"</code> in the 3D case). This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient q is zero. The Poisson Solver verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, the Poisson Solver computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the <code>?_Helmholtz_2D/?_Helmholtz_3D</code> routine informs you that the solution may not exist in a classical sense (up to rounding errors). <li data-bbox="420 737 1458 877">• In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the Cartesian case with pure Neumann boundary conditions. Here the Poisson Solver computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors. <p>The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.</p>
<p><code>ipar[13]-1</code> through <code>ipar[14]-1</code></p>	<p>In the Cartesian case, contain the spectrum of the one-dimensional (1D) problem along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the spectrum of the 1D problem along the ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
<p><code>ipar[15]-1</code> through <code>ipar[16]-1</code></p>	<p>In the Cartesian case, contain the spectrum of the 1D problem along the y-axis after a call to the <code>?_commit_Helmholtz_3D</code> routine. These elements are used only in the 3D case.</p> <p>In the spherical case, contains the spherical weights after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
<p><code>ipar[17]-1</code> through <code>ipar[18]-1</code></p>	<p>Take the values of the (staggered) sine/cosine in the mesh points:</p> <ul style="list-style-type: none"> <li data-bbox="420 1430 1219 1486">• along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine for a Cartesian solver <li data-bbox="420 1493 1422 1549">• along the ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine for a spherical solver.
<p><code>ipar[19]-1</code> through <code>ipar[20]-1</code></p>	<p>Take the values of the (staggered) sine/cosine in the mesh points:</p> <ul style="list-style-type: none"> <li data-bbox="420 1633 1349 1690">• along the y-axis after a call to the <code>?_commit_Helmholtz_3D</code> routine for a Cartesian 3D solver <li data-bbox="420 1696 1357 1753">• along the ϕ-axis after a call to the <code>?_commit_sph_p</code> routine for a spherical periodic solver.
	<p>These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.</p>

Index	Description
<i>ipar</i> [23]-1 through <i>ipar</i> [24]-1	Take the values of the (staggered) sine/cosine in the mesh points along the <i>x</i> -axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. These elements are used only in the periodic Cartesian case.
<i>ipar</i> [25]-1 through <i>ipar</i> [26]-1	Take the values of the (staggered) sine/cosine in the mesh points along the <i>x</i> -axis after a call to the <code>?_commit_Helmholtz_3D</code> routine. These elements are used only in the periodic 3D Cartesian case.

NOTE

You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the Poisson Solver interface enables you to skip calls to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of the *ipar*, *dpar*, and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or incorrect results. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to Poisson Solver routines, you should have considerable experience in using the Poisson Solver interface and good understanding of the solution process, as well as elements contained in the *ipar*, *spar*, and *dpar* arrays and dependencies between values of these elements.

In rare occurrences when you fail in tuning parameters for the Fast Helmholtz Solver, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar*, and *spar* arrays unless it is necessary.

Parameters That Define Boundary Conditions

Poisson Solver routines for the Cartesian solver use the following common parameters to define the boundary conditions.

Parameters to Define Boundary Conditions for the Cartesian Solver

Parameter	Description
<i>bd_ax</i>	double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_2D/d_Helmholtz_3D</code> , float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code> . Contains values of the boundary condition on the leftmost boundary of the domain along the <i>x</i> -axis.

Parameter	Description
	<ul style="list-style-type: none"> 2D problem: the size of the array is n_y+1. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[0]$ is 'D'): values of the function $G(ax, y_j), j=0, \dots, n_y$. Neumann boundary condition (value of $BCtype[0]$ is 'N'): values of the function $g(ax, y_j), j=0, \dots, n_y$. The value corresponding to the index j is placed in $bd_ax[j]$. 3D problem: the size of the array is $(n_y+1)*(n_z+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[0]$ is 'D'): values of the function $G(ax, y_j, z_k), j=0, \dots, n_y, k=0, \dots, n_z$. Neumann boundary condition (value of $BCtype[0]$ is 'N'): the values of the function $g(ax, y_j, z_k), j=0, \dots, n_y, k=0, \dots, n_z$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in $bd_ax[j+k*(n_y+1)]$. <p>For periodic boundary conditions (the value of $BCtype[0]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_bx</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_2D/d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code>.</p> <p>Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is n_y+1. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[1]$ is 'D'): values of the function $G(bx, y_j), j=0, \dots, n_y$. Neumann boundary condition (value of $BCtype[1]$ is 'N'): values of the function $g(bx, y_j), j=0, \dots, n_y$. The value corresponding to the index j is placed in $bd_bx[j]$. 3D problem: the size of the array is $(n_y+1)*(n_z+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[1]$ is 'D'): values of the function $G(bx, y_j, z_k), j=0, \dots, n_y, k=0, \dots, n_z$. Neumann boundary condition (value of $BCtype[1]$ is 'N'): values of the function $g(bx, y_j, z_k), j=0, \dots, n_y, k=0, \dots, n_z$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in $bd_bx[j+k*(n_y+1)]$. <p>For periodic boundary conditions (the value of $BCtype[1]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_ay</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_2D/d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code>.</p>

Parameter	Description
	<p>Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[2]$ is 'D'): values of the function $G(x_i, ay)$, $i=0, \dots, nx$. Neumann boundary condition (value of $BCtype[2]$ is 'N'): values of the function $g(x_i, ay)$, $i=0, \dots, nx$. <p>The value corresponding to the index i is placed in $bd_ay[i]$.</p> 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[2]$ is 'D'): values of the function $G(x_i, ay, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. Neumann boundary condition (value of $BCtype[2]$ is 'N'): values of the function $g(x_i, ay, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (i, k) is placed in $bd_ay[i+k*(nx+1)]$.</p> <p>For periodic boundary conditions (the value of $BCtype[2]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_by</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_2D/d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code>.</p> <p>Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[3]$ is 'D'): values of the function $G(x_i, by)$, $i=0, \dots, nx$. Neumann boundary condition (value of $BCtype[3]$ is 'N'): values of the function $g(x_i, by)$, $i=0, \dots, nx$. <p>The value corresponding to the index i is placed in $bd_by[i]$.</p> 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[3]$ is 'D'): values of the function $G(x_i, by, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. Neumann boundary condition (value of $BCtype[3]$ is 'N'): values of the function $g(x_i, by, z_k)$, $i=0, \dots, nx, k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (i, k) is placed in $bd_by[i+k*(nx+1)]$.</p> <p>For periodic boundary conditions (the value of $BCtype[3]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_az</code>	<p>double* for <code>d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_3D</code>.</p>

Parameter	Description
	<p>Used only by <code>?_commit_Helmholtz_3D</code> and <code>?_Helmholtz_3D</code>. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[4]</code> is 'D'): values of the function $G(x_i, y_j, az)$, $i=0, \dots, nx, j=0, \dots, ny$. Neumann boundary condition (value of <code>BCtype[4]</code> is 'N'), values of the function $g(x_i, y_j, az)$, $i=0, \dots, nx, j=0, \dots, ny$. <p>The values are packed in the array so that the value corresponding to indices (i, j) is placed in <code>bd_az[i+j*(nx+1)]</code>.</p> <p>For periodic boundary conditions (the value of <code>BCtype[4]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_bz</code>	<p><code>double*</code> for <code>d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_3D</code>, <code>float*</code> for <code>s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_3D</code>.</p> <p>Used only by <code>?_commit_Helmholtz_3D</code> and <code>?_Helmholtz_3D</code>. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[5]</code> is 'D'): values of the function $G(x_i, y_j, bz)$, $i=0, \dots, nx, j=0, \dots, ny$. Neumann boundary condition (value of <code>BCtype[5]</code> is 'N'): values of the function $g(x_i, y_j, bz)$, $i=0, \dots, nx, j=0, \dots, ny$. <p>The values are packed in the array so that the value corresponding to indices (i, j) is placed in <code>bd_bz[i+j*(nx+1)]</code>.</p> <p>For periodic boundary conditions (the value of <code>BCtype[5]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>

See Also

[_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#)
[?_Helmholtz_2D/?_Helmholtz_3D](#)

Implementation Details

Several aspects of the Intel MKL Poisson Solver interface are platform-specific and language-specific. To promote portability of the Intel MKL Poisson Solver interface across platforms and ease of use across different languages, Intel MKL provides you with the Poisson Solver language-specific header file to include in your code:

- `mkl_poisson.f90`, to be used together with `mkl_dfti.f90`.

NOTE

- Intel MKL Poisson Solver interface supports Fortran versions starting with Fortran 90.
- Use of the Intel MKL Poisson Solver software without including the above language-specific header files is not supported.

Header File

The header file defines the following function prototypes for the Cartesian solver:

```
SUBROUTINE D_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, DPAR, STAT)
  USE MKL_DFTI

  INTEGER NX, NY, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AX, BX, AY, BY, Q
  DOUBLE PRECISION DPAR(*)
  CHARACTER(4) BCTYPE
END SUBROUTINE
```

```
SUBROUTINE D_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE
```

```
SUBROUTINE D_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE
```

```
SUBROUTINE S_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, SPAR, STAT)
  USE MKL_DFTI

  INTEGER NX, NY, STAT
  INTEGER IPAR(*)
  REAL AX, BX, AY, BY, Q
  REAL SPAR(*)
  CHARACTER(4) BCTYPE
END SUBROUTINE
```

```
SUBROUTINE S_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL F(IPAR(11)+1,*)
  REAL SPAR(*)
  REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE
```

```

SUBROUTINE S_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL F(IPAR(11)+1,*)
  REAL SPAR(*)
  REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_2D (XHANDLE, IPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE D_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, DPAR, STAT)
  USE MKL_DFTI

  INTEGER NX, NY, NZ, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AX, BX, AY, BY, AZ, BZ, Q
  DOUBLE PRECISION DPAR(*)
  CHARACTER(6) BCTYPE
END SUBROUTINE

SUBROUTINE D_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
DPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1, IPAR(12)+1, *)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(IPAR(12)+1, *), BD_BX(IPAR(12)+1, *), BD_AY(IPAR(11)+1, *)
  DOUBLE PRECISION BD_BY(IPAR(11)+1, *), BD_AZ(IPAR(11)+1, *), BD_BZ(IPAR(11)+1, *)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE D_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR, DPAR,
STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1, IPAR(12)+1, *)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(IPAR(12)+1, *), BD_BX(IPAR(12)+1, *), BD_AY(IPAR(11)+1, *)
  DOUBLE PRECISION BD_BY(IPAR(11)+1, *), BD_AZ(IPAR(11)+1, *), BD_BZ(IPAR(11)+1, *)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

```

```

SUBROUTINE S_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, SPAR, STAT)
  USE MKL_DFTI

  INTEGER NX, NY, NZ, STAT
  INTEGER IPAR(*)
  REAL AX, BX, AY, BY, AZ, BZ, Q
  REAL SPAR(*)
  CHARACTER(6) BCTYPE
END SUBROUTINE

SUBROUTINE S_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
SPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL F(IPAR(11)+1,IPAR(12)+1,*)
  REAL SPAR(*)
  REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
  REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE S_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR, SPAR,
STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL F(IPAR(11)+1,IPAR(12)+1,*)
  REAL SPAR(*)
  REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
  REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_3D (XHANDLE, YHANDLE, IPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

```

The header file defines the following function prototypes for the spherical solver:

```

SUBROUTINE D_INIT_SPH_P (AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
  USE MKL_DFTI

  INTEGER NP, NT, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AP,BP,AT,BT,Q
  DOUBLE PRECISION DPAR(*)
END SUBROUTINE

```

```

SUBROUTINE D_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE D_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE S_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER NP, NT, STAT
  INTEGER IPAR(*)
  REAL AP,BP,AT,BT,Q
  REAL SPAR(*)
END SUBROUTINE

SUBROUTINE S_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE S_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)

```

```

    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_S, HANDLE_C
END SUBROUTINE

SUBROUTINE D_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER NP, NT, STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION AP,BP,AT,BT,Q
    DOUBLE PRECISION DPAR(*)
END SUBROUTINE

SUBROUTINE D_COMMIT_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION DPAR(*)
    DOUBLE PRECISION F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE D_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    DOUBLE PRECISION DPAR(*)
    DOUBLE PRECISION F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE S_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
    USE MKL_DFTI

    INTEGER NP, NT, STAT
    INTEGER IPAR(*)
    REAL AP,BP,AT,BT,Q
    REAL SPAR(*)
END SUBROUTINE

SUBROUTINE S_COMMIT_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    REAL SPAR(*)
    REAL F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE S_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
    USE MKL_DFTI

```

```

    INTEGER STAT
    INTEGER IPAR(*)
    REAL SPAR(*)
    REAL F(IPAR(11)+1,*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE FREE_SPH_NP(HANDLE,IPAR,STAT)
    USE MKL_DFTI

    INTEGER STAT
    INTEGER IPAR(*)
    TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

```

Fortran specifics of the Poisson Solver routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran](#) section.

Calling PDE Support Routines from Fortran

The calling interface for all the Intel MKL TT and Poisson Solver routines is designed to be easily used in C. However, you can invoke each TT or Poisson Solver routine directly from Fortran 90 or higher if you are familiar with the inter-language calling conventions of your platform.

The TT or Poisson Solver interface cannot be invoked from FORTRAN 77 due to restrictions imposed by the use of the Intel MKL FFT interface.

The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from C to Fortran, and how C external names are decorated on the platform.

To promote portability and relieve you of dealing with the calling conventions specifics, the Fortran header file `mkl_trig_transforms.f90` for TT routines and `mkl_poisson.f90` for Poisson Solver routines, used together with `mkl_dfti.f90`, declare a set of macros and introduce type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks natural in Fortran.

For example, consider a hypothetical library routine, `foo`, which takes a double-precision vector of length n . C users access such a function as follows:

```

MKL_INT n;
double *x;
...
foo(x, &n);

```

As noted above, to invoke `foo`, Fortran users would need to know what Fortran data types correspond to C types `MKL_INT` and `double` (or `float` for single-precision), what argument-passing mechanism the C compiler uses and what, if any, name decoration is performed by the C compiler when generating the external symbol `foo`. However, with the Fortran header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` included, the invocation of `foo` within a Fortran program will look as follows for the LP64 interface (for the ILP64 interface, `INTEGER*8` type will be used instead of `INTEGER*4`):

- For TT interface,

```

use mkl_dfti
use mkl_trig_transforms
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
...
CALL FOO(x,n)

```

- For Poisson Solver interface,

```
use mkl_dfti
use mkl_poisson
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
...
CALL FOO(x,n)
```

Note that in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` provide a definition for the subroutine `FOO`. To ease the use of Poisson Solver or TT routines in Fortran, the general approach of providing Fortran definitions of names is used throughout the libraries. Specifically, if a name from a Poisson Solver or TT interface is documented as having the C-specific name `foo`, then the Fortran header files provide an appropriate Fortran language type definition `FOO`.

One of the key differences between Fortran and C is the language argument-passing mechanism: C programs use pass-by-value semantics and Fortran programs use pass-by-reference semantics. The Fortran headers ensure proper treatment of this difference. In particular, in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` hide the difference by defining a macro `FOO` that takes the address of the appropriate arguments.

See Also

[C Datatypes Specific to Intel MKL](#)

Nonlinear Optimization Problem Solvers

13

Intel® Math Kernel Library (Intel® MKL) provides tools for solving nonlinear least squares problems using the Trust-Region (TR) algorithms. The solver routines are grouped according to their purpose as follows:

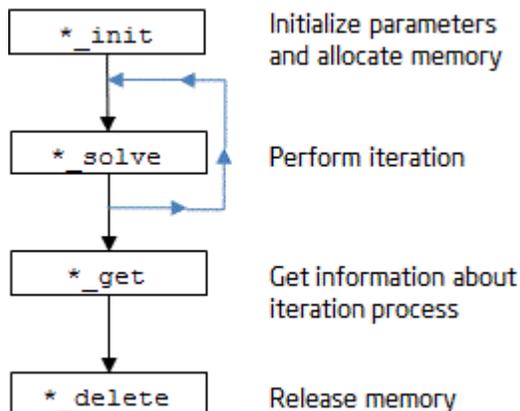
- [Nonlinear Least Squares Problem without Constraints](#)
- [Nonlinear Least Squares Problem with Linear \(Boundary\) Constraints](#)
- [Jacobian Matrix Calculation Routines](#)

For more information on the key concepts required to understand the use of the Intel MKL nonlinear least squares problem solver routines, see [\[Conn00\]](#).

Organization and Implementation

The Intel MKL solver routines for nonlinear least squares problems use reverse communication interfaces (RCI). That means you need to provide the solver with information required for the iteration process, for example, the corresponding Jacobian matrix, or values of the objective function. RCI removes the dependency of the solver on specific implementation of the operations. However, it does require that you organize a computational loop.

Typical order for invoking RCI solver routines



The nonlinear least squares problem solver routines, or Trust-Region (TR) solvers, are implemented with threading support. You can manage the threads using [Threading Control Functions](#). The TR solvers use BLAS and LAPACK routines, and offer the same parallelism as those domains. The `?jacobi` and `?jacobix` routines of Jacobian matrix calculations are parallel. These routines (`?jacobi` and `?jacobix`) make calls to the user-supplied functions with different x parameters for multiple threads.

Memory Allocation and Handles

To make the TR solver routines easy to use, you are not required to allocate temporary working storage. The solver allocates any required memory. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a data object called a *handle*. Each TR solver routine creates, uses, or deletes a handle. To declare a handle, include `mkl_rci.fi`.

For a program using compilers that support eight byte integers, declare a handle as:

```
INCLUDE "mkl_rci.fi"
INTEGER*8 handle
```

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Routine Naming Conventions

The TR routine names have the following structure:

```
<character><name>_<action>( )
```

where

- *<character>* indicates the data type:

s real, single precision

d real, double precision

- *<name>* indicates the task type:

trnlsp nonlinear least squares problem without constraints

trnlspbc nonlinear least squares problem with boundary constraints

jacobi computation of the Jacobian matrix using central differences

- *<action>* indicates an action on the task:

init initializes the solver

check checks correctness of the input parameters

solve solves the problem

get retrieves the number of iterations, the stop criterion, the initial residual, and the final residual

delete releases the allocated data

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in \mathbb{R}^n} \|F(x)\|_2^2 = \min_{x \in \mathbb{R}^n} \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n,$$

where

$F(x) : R^n \rightarrow R^m$ is a twice differentiable function in R^n .

Solving a nonlinear least squares problem means searching for the best approximation to the vector y with the model function $f_i(x)$ and nonlinear variables x . The best approximation means that the sum of squares of residuals $y_i - f_i(x)$ is the minimum.

See usage examples in the `examples\solverf\source` folder of your Intel MKL directory. Specifically, see `ex_nlsqp_f.f`.

RCI TR Routines

Routine Name	Operation
<code>?trnlsq_init</code>	Initializes the solver.
<code>?trnlsq_check</code>	Checks correctness of the input parameters.
<code>?trnlsq_solve</code>	Solves a nonlinear least squares problem using the Trust-Region algorithm.
<code>?trnlsq_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>?trnlsq_delete</code>	Releases allocated data.

?trnlsq_init

Initializes the solver of a nonlinear least squares problem.

Syntax

```
res = strnlsq_init(handle, n, m, x, eps, iter1, iter2, rs)
```

```
res = dtrnlsq_init(handle, n, m, x, eps, iter1, iter2, rs)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?trnlsq_init` routine initializes the solver.

After initialization, all subsequent invocations of the `?trnlsq_solve` routine should use the values of the `handle` returned by `?trnlsq_init`.

The `eps` array contains a number indicating the stopping criteria:

<code>eps</code> Value	Description
1	$\Delta < eps(1)$
2	$\ F(x)\ _2 < eps(2)$
3	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
4	$\ s\ _2 < eps(4)$
5	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$

<i>eps</i> Value	Description
6	The trial step precision. If $eps(6) = 0$, then the trial step meets the required precision ($\leq 1.0 \cdot 10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>n</i>	INTEGER. Length of x .
<i>m</i>	INTEGER. Length of $F(x)$.
<i>x</i>	REAL for <code>strnlsp_init</code> DOUBLE PRECISION for <code>dtrnlsp_init</code> Array of size n . Initial guess.
<i>eps</i>	REAL for <code>strnlsp_init</code> DOUBLE PRECISION for <code>dtrnlsp_init</code> Array of size 6; contains stopping criteria. See the values in the Description section.
<i>iter1</i>	INTEGER. Specifies the maximum number of iterations.
<i>iter2</i>	INTEGER. Specifies the maximum number of iterations of trial step calculation.
<i>rs</i>	REAL for <code>strnlsp_init</code> DOUBLE PRECISION for <code>dtrnlsp_init</code> Definition of initial size of the trust region (boundary of the trial step). The recommend minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <i>rs</i> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<i>handle</i>	Type INTEGER*8.
<i>res</i>	INTEGER. Indicates task completion status. <ul style="list-style-type: none"> • $res = TR_SUCCESS$ - the routine completed the task normally. • $res = TR_INVALID_OPTION$ - there was an error in the input parameters. • $res = TR_OUT_OF_MEMORY$ - there was a memory error. <p><code>TR_SUCCESS</code>, <code>TR_INVALID_OPTION</code>, and <code>TR_OUT_OF_MEMORY</code> are defined in the <code>mkl_rci.fi</code> include file.</p>

See Also[?trnlsp_solve](#)**?trnlsp_check**

Checks the correctness of *handle* and arrays containing Jacobian matrix, objective function, and stopping criteria.

Syntax

```
res = strnlsp_check(handle, n, m, fjac, fvec, eps, info)
```

```
res = dtrnlsp_check(handle, n, m, fjac, fvec, eps, info)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?trnlsp_check` routine checks the arrays passed into the solver as input parameters. If an array contains any `INF` or `NaN` values, the routine sets the flag in output array *info* (see the description of the values returned in the Output Parameters section for the *info* array).

Input Parameters

<i>handle</i>	Type <code>INTEGER*8</code> .
<i>n</i>	<code>INTEGER</code> . Length of <i>x</i> .
<i>m</i>	<code>INTEGER</code> . Length of $F(x)$.
<i>fjac</i>	<code>REAL</code> for <code>strnlsp_check</code> <code>DOUBLE PRECISION</code> for <code>dtrnlsp_check</code> Array of size <i>m</i> by <i>n</i> . Contains the Jacobian matrix of the function.
<i>fvec</i>	<code>REAL</code> for <code>strnlsp_check</code> <code>DOUBLE PRECISION</code> for <code>dtrnlsp_check</code> Array of size <i>m</i> . Contains the function values at <i>x</i> , where $fvec(i) = (y_i - f_i(x))$.
<i>eps</i>	<code>REAL</code> for <code>strnlsp_check</code> <code>DOUBLE PRECISION</code> for <code>dtrnlsp_check</code> Array of size 6; contains stopping criteria. See the values in the Description section of the ?trnlsp_init .

Output Parameters

<i>info</i>	<code>INTEGER</code> Array of size 6. Results of input parameter checking:
-------------	--

Parameter	Used for	Value	Description
<i>info(1)</i>	Flags for <i>handle</i>	0	The handle is valid.
		1	The handle is not allocated.
<i>info(2)</i>	Flags for <i>fjac</i>	0	The <i>fjac</i> array is valid.
		1	The <i>fjac</i> array is not allocated
		2	The <i>fjac</i> array contains NaN.
		3	The <i>fjac</i> array contains Inf.
<i>info(3)</i>	Flags for <i>fvec</i>	0	The <i>fvec</i> array is valid.
		1	The <i>fvec</i> array is not allocated
		2	The <i>fvec</i> array contains NaN.
		3	The <i>fvec</i> array contains Inf.
<i>info(4)</i>	Flags for <i>eps</i>	0	The <i>eps</i> array is valid.
		1	The <i>eps</i> array is not allocated
		2	The <i>eps</i> array contains NaN.
		3	The <i>eps</i> array contains Inf.
		4	The <i>eps</i> array contains a value less than or equal to zero.

res

INTEGER. Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the `mkl_rci.fi` include file.

?trnlsp_solve

Solves a nonlinear least squares problem using the TR algorithm.

Syntax

```
res = strnlsp_solve(handle, fvec, fjac, RCI_Request)
```

```
res = dtrnlsp_solve(handle, fvec, fjac, RCI_Request)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The ?trnlsp_solve routine uses the TR algorithm to solve nonlinear least squares problems.

The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \|F(x)\|_2^2 = \min_{x \in \mathbb{R}^n} \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n,$$

where

- $F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- $m \geq n$

From a current point $x_{current}$, the algorithm uses the trust-region approach:

$$\min_{x \in \mathbb{R}^n} \|F(x_{current}) + J(x_{current})(x_{new} - x_{current})\|_2^2 \quad \text{subject to} \quad \|x_{new} - x_{current}\| \leq \Delta_{current}$$

to get $x_{new} = x_{current} + s$ that satisfies

$$\min_{s \in \mathbb{R}^n} \|J^T(x)J(x)s + J^T F(x)\|_2^2$$

where

- $J(x)$ is the Jacobian matrix
- s is the trial step
- $\|s\|_2 \leq \Delta_{current}$

The *RCI_Request* parameter provides additional information:

<i>RCI_Request</i> Value	Description
2	Request to calculate the Jacobian matrix and put the result into <i>fjac</i>
1	Request to recalculate the function at vector x and put the result into <i>fvec</i>
0	One successful iteration step on the current trust-region radius (that does not mean that the value of x has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < eps(1)$
-3	$\ F(x)\ _2 < eps(2)$
-4	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
-5	$\ s\ _2 < eps(4)$
-6	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>handle</i>	Type INTEGER*8.
<i>fvec</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size <i>m</i> . Contains the function values at <i>X</i> , where $fvec(i) = (y_i - f_i(x))$.
<i>fjac</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size <i>m</i> by <i>n</i> . Contains the Jacobian matrix of the function.

Output Parameters

<i>fvec</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size <i>m</i> . Updated function evaluated at <i>x</i> .
<i>RCI_Request</i>	INTEGER. Informs about the task stage. See the Description section for the parameter values and their meaning.
<i>res</i>	INTEGER. Indicates the task completion. <i>res</i> = TR_SUCCESS - the routine completed the task normally. TR_SUCCESS is defined in the <code>mkl_rci.fi</code> include file.

?trnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

```
res = strnlsp_get(handle, iter, st_cr, r1, r2)
res = dtrnlsp_get(handle, iter, st_cr, r1, r2)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The initial residual is the value of the functional $(\|y - f(x)\|)$ of the initial *x* values provided by the user.

The final residual is the value of the functional $(\|y - f(x)\|)$ of the final *x* resulting from the algorithm operation.

The *st_cr* parameter contains a number indicating the stop criterion:

<i>st_cr</i> Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps(1)$
3	$\ F(x)\ _2 < eps(2)$
4	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
5	$\ s\ _2 < eps(4)$
6	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type INTEGER*8.

Output Parameters

iter INTEGER. Contains the current number of iterations.

st_cr INTEGER. Contains the stop criterion.
See the Description section for the parameter values and their meanings.

r1 REAL for `strnlsp_get`
DOUBLE PRECISION for `dtrnlsp_get`
Contains the residual, $(\|y - f(x)\|)$ given the initial x .

r2 REAL for `strnlsp_get`
DOUBLE PRECISION for `dtrnlsp_get`
Contains the final residual, that is, the value of the functional $(\|y - f(x)\|)$ of the final x resulting from the algorithm operation.

res INTEGER. Indicates the task completion.
res = TR_SUCCESS - the routine completed the task normally.
TR_SUCCESS is defined in the `mkl_rci.fi` include file.

?trnlsp_delete

Releases allocated data.

Syntax

res = `strnlsp_delete(handle)`

Syntax

```
res = strnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
res = dtrnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
```

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlspbc_init routine initializes the solver.

After initialization all subsequent invocations of the ?trnlspbc_solve routine should use the values of the handle returned by ?trnlspbc_init.

The eps array contains a number indicating the stopping criteria:

eps Value	Description
1	$\Delta < eps(1)$
2	$\ F(x)\ _2 < eps(2)$
3	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
4	$\ s\ _2 < eps(4)$
5	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$
6	The trial step precision. If $eps(6) = 0$, then the trial step meets the required precision ($\leq 1.0 \cdot 10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

n INTEGER. Length of x .

m INTEGER. Length of $F(x)$.

x REAL for strnlspbc_init
DOUBLE PRECISION for dtrnlspbc_init
Array of size n . Initial guess.

LW REAL for strnlspbc_init
DOUBLE PRECISION for dtrnlspbc_init
Array of size n .
Contains low bounds for x ($lw_i < x_i$).

UP REAL for strnlspbc_init

	DOUBLE PRECISION for <code>dtrnlspbc_init</code>
	Array of size n .
	Contains upper bounds for x ($up_i > x_i$).
<code>eps</code>	REAL for <code>strnlspbc_init</code>
	DOUBLE PRECISION for <code>dtrnlspbc_init</code>
	Array of size 6; contains stopping criteria. See the values in the Description section.
<code>iter1</code>	INTEGER. Specifies the maximum number of iterations.
<code>iter2</code>	INTEGER. Specifies the maximum number of iterations of trial step calculation.
<code>rs</code>	REAL for <code>strnlspbc_init</code>
	DOUBLE PRECISION for <code>dtrnlspbc_init</code>
	Definition of initial size of the trust region (boundary of the trial step). The recommended minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <code>rs</code> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<code>handle</code>	Type INTEGER*8.
<code>res</code>	INTEGER. Informs about the task completion. <ul style="list-style-type: none"> • <code>res = TR_SUCCESS</code> - the routine completed the task normally. • <code>res = TR_INVALID_OPTION</code> - there was an error in the input parameters. • <code>res = TR_OUT_OF_MEMORY</code> - there was a memory error. <p><code>TR_SUCCESS</code>, <code>TR_INVALID_OPTION</code>, and <code>TR_OUT_OF_MEMORY</code> are defined in the <code>mkl_rci.fi</code> include file.</p>

?trnlspbc_check

Checks the correctness of handle and arrays containing Jacobian matrix, objective function, lower and upper bounds, and stopping criteria.

Syntax

```
res = strnlspbc_check(handle, n, m, fjac, fvec, LW, UP, eps, info)
res = dtrnlspbc_check(handle, n, m, fjac, fvec, LW, UP, eps, info)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Parameter	Used for	Value	Description
		1	The handle is not allocated.
<i>info(2)</i>	Flags for <i>fjac</i>	0	The <i>fjac</i> array is valid.
		1	The <i>fjac</i> array is not allocated
		2	The <i>fjac</i> array contains NaN.
		3	The <i>fjac</i> array contains Inf.
<i>info(3)</i>	Flags for <i>fvec</i>	0	The <i>fvec</i> array is valid.
		1	The <i>fvec</i> array is not allocated
		2	The <i>fvec</i> array contains NaN.
		3	The <i>fvec</i> array contains Inf.
<i>info(4)</i>	Flags for <i>LW</i>	0	The <i>LW</i> array is valid.
		1	The <i>LW</i> array is not allocated
		2	The <i>LW</i> array contains NaN.
		3	The <i>LW</i> array contains Inf.
		4	The lower bound is greater than the upper bound.
<i>info(5)</i>	Flags for <i>up</i>	0	The <i>up</i> array is valid.
		1	The <i>up</i> array is not allocated
		2	The <i>up</i> array contains NaN.
		3	The <i>up</i> array contains Inf.
		4	The upper bound is less than the lower bound.
<i>info(6)</i>	Flags for <i>eps</i>	0	The <i>eps</i> array is valid.
		1	The <i>eps</i> array is not allocated
		2	The <i>eps</i> array contains NaN.
		3	The <i>eps</i> array contains Inf.
		4	The <i>eps</i> array contains a value less than or equal to zero.

res

INTEGER. Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the `mkl_rci.fi` include file.

?trnlspbc_solve

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

```
res = strnlspbc_solve(handle, fvec, fjac, RCI_Request)
```

```
res = dtrnlspbc_solve(handle, fvec, fjac, RCI_Request)
```

Include Files

- Fortran: mkl_rci.fi, mkl_rci.f90

Description

The ?trnlspbc_solve routine, based on RCI, uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \|F(x)\|_2^2 = \min_{x \in \mathbb{R}^n} \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n$$

where

$$l_i \leq x_i \leq u_i$$

$$i = 1, \dots, n.$$

The *RCI_Request* parameter provides additional information:

<i>RCI_Request</i> Value	Description
2	Request to calculate the Jacobian matrix and put the result into <i>fjac</i>
1	Request to recalculate the function at vector <i>x</i> and put the result into <i>fvec</i>
0	One successful iteration step on the current trust-region radius (that does not mean that the value of <i>x</i> has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < eps(1)$
-3	$\ F(x)\ _2 < eps(2)$
-4	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
-5	$\ s\ _2 < eps(4)$
-6	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>handle</i>	Type INTEGER*8.
<i>fvec</i>	REAL for <code>strnlsppbc_solve</code> DOUBLE PRECISION for <code>dtrnlsppbc_solve</code> Array of size <i>m</i> . Contains the function values at X, where $fvec(i) = (y_i - f_i(x))$.
<i>fjac</i>	REAL for <code>strnlsppbc_solve</code> DOUBLE PRECISION for <code>dtrnlsppbc_solve</code> Array of size <i>m</i> by <i>n</i> . Contains the Jacobian matrix of the function.

Output Parameters

<i>fvec</i>	REAL for <code>strnlsppbc_solve</code> DOUBLE PRECISION for <code>dtrnlsppbc_solve</code> Array of size <i>m</i> . Updated function evaluated at x.
<i>RCI_Request</i>	INTEGER. Informs about the task stage. See the Description section for the parameter values and their meaning.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = TR_SUCCESS means the routine completed the task normally. TR_SUCCESS is defined in the <code>mkl_rci.fi</code> include file.

?trnlsppbc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

```
res = strnlsppbc_get(handle, iter, st_cr, r1, r2)
res = dtrnlsppbc_get(handle, iter, st_cr, r1, r2)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The *st_cr* parameter contains a number indicating the stop criterion:

<i>st_cr</i> Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps(1)$

<i>st_cr</i> Value	Description
3	$\ F(x)\ _2 < eps(2)$
4	The Jacobian matrix is singular. $\ J(x)_{(1:m,j)}\ _2 < eps(3), j = 1, \dots, n$
5	$\ s\ _2 < eps(4)$
6	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < eps(5)$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type INTEGER*8.

Output Parameters

iter INTEGER. Contains the current number of iterations.

st_cr INTEGER. Contains the stop criterion.
See the Description section for the parameter values and their meanings.

r1 REAL for `strnlspbc_get`
DOUBLE PRECISION for `dtrnlspbc_get`
Contains the residual, $(\|y - f(x)\|)$ given the initial x .

r2 REAL for `strnlspbc_get`
DOUBLE PRECISION for `dtrnlspbc_get`
Contains the final residual, that is, the value of the function $(\|y - f(x)\|)$ of the final x resulting from the algorithm operation.

res INTEGER. Informs about the task completion.
res = TR_SUCCESS - the routine completed the task normally.
TR_SUCCESS is defined in the `mkl_rci.fi` include file.

?trnlspbc_delete

Releases allocated data.

Syntax

res = `strnlspbc_delete(handle)`

res = `dtrnlspbc_delete(handle)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?trnlspbc_delete` routine releases all memory allocated for the handle.

NOTE

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_free_buffers`.

Input Parameters

`handle` Type `INTEGER*8`.

Output Parameters

`res` `INTEGER`. Informs about the task completion.
`res = TR_SUCCESS` means the routine completed the task normally.
`TR_SUCCESS` is defined in the `mkl_rci.fi` include file.

Jacobian Matrix Calculation Routines

This section describes routines that compute the Jacobian matrix using the central difference algorithm. Jacobian matrix calculation is required to solve a nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobian matrix have the "Black-Box" interfaces, where you pass the objective function via parameters. Your objective function must have a fixed interface.

Jacobian Matrix Calculation Routines

Routine Name	Operation
<code>?jacobi_init</code>	Initializes the solver.
<code>?jacobi_solve</code>	Computes the Jacobian matrix of the function on the basis of RCI using the central difference algorithm.
<code>?jacobi_delete</code>	Removes data.
<code>?jacobi</code>	Computes the Jacobian matrix of the <code>fcn</code> function using the central difference algorithm.
<code>?jacobix</code>	Presents an alternative interface for the <code>?jacobi</code> function enabling you to pass additional data into the objective function.

`?jacobi_init`

Initializes the solver for Jacobian calculations.

Syntax

```
res = sjacobi_init(handle, n, m, x, fjac, eps)
res = djacobi_init(handle, n, m, x, fjac, eps)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The routine initializes the solver.

Input Parameters

<code>n</code>	INTEGER. Length of x .
<code>m</code>	INTEGER. Length of F .
<code>x</code>	REAL for <code>sjacobi_init</code> DOUBLE PRECISION for <code>djacobi_init</code> Array of size n . Vector, at which the function is evaluated.
<code>eps</code>	REAL for <code>sjacobi_init</code> DOUBLE PRECISION for <code>djacobi_init</code> Precision of the Jacobian matrix calculation.
<code>fjac</code>	REAL for <code>sjacobi_init</code> DOUBLE PRECISION for <code>djacobi_init</code> Array of size m by n . Contains the Jacobian matrix of the function.

Output Parameters

<code>handle</code>	Data object of the INTEGER*8.
<code>res</code>	INTEGER. Indicates task completion status. <ul style="list-style-type: none"> • <code>res = TR_SUCCESS</code> - the routine completed the task normally. • <code>res = TR_INVALID_OPTION</code> - there was an error in the input parameters. • <code>res = TR_OUT_OF_MEMORY</code> - there was a memory error. <p><code>TR_SUCCESS</code>, <code>TR_INVALID_OPTION</code>, and <code>TR_OUT_OF_MEMORY</code> are defined in the <code>mkl_rci.fi</code> include file.</p>

?jacobi_solve

Computes the Jacobian matrix of the function using RCI and the central difference algorithm.

Syntax

```
res = sjacobi_solve(handle, f1, f2, RCI_Request)
```

```
res = djacobi_solve(handle, f1, f2, RCI_Request)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?jacobi_solve` routine computes the Jacobian matrix of the function using RCI and the central difference algorithm.

See usage examples in the `examples\solverf\source` folder of your Intel MKL directory. Specifically, see `sjacobi_rci_f.f` and `djacobi_rci_f.f`.

Input Parameters

handle Type INTEGER*8.

RCI_Request INTEGER. Set to 0 before the first call to `?jacobi_solve`.

Output Parameters

f1 REAL for `sjacobi_solve`
DOUBLE PRECISION for `djacobi_solve`
Contains the updated function values at $x + eps$.

f2 REAL for `sjacobi_solve`
DOUBLE PRECISION for `djacobi_solve`
Array of size m . Contains the updated function values at $x - eps$.

RCI_Request Provides information about the task completion. When equal to 0, the task has completed successfully.
RCI_Request= 1 indicates that you should compute the function values at the current x point and put the results into *f1*.
RCI_Request= 2 indicates that you should compute the function values at the current x point and put the results into *f2*.

res INTEGER. Indicates the task completion status.

- res* = TR_SUCCESS - the routine completed the task normally.
- res* = TR_INVALID_OPTION - there was an error in the input parameters.

TR_SUCCESS and TR_INVALID_OPTION are defined in the `mkl_rci.fi` include file.

See Also

`?jacobi_init`

?jacobi_delete

Releases allocated data.

Syntax

`res = sjacobi_delete(handle)`

`res = djacobi_delete(handle)`

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?jacobi_delete` routine releases all memory allocated for the handle.

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_free_buffers`.

Input Parameters

`handle` Type `INTEGER*8`.

Output Parameters

`res` `INTEGER`. Informs about the task completion.
`res = TR_SUCCESS` means the routine completed the task normally.
`TR_SUCCESS` is defined in the `mkl_rci.fi` include file.

?jacobi

Computes the Jacobian matrix of the objective function using the central difference algorithm.

Syntax

```
res = sjacobi(fcn, n, m, fjac, x, eps)
```

```
res = djacobi(fcn, n, m, fjac, x, eps)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?jacobi` routine computes the Jacobian matrix for function `fcn` using the central difference algorithm. This routine has a "Black-Box" interface, where you input the objective function via parameters. Your objective function must have a fixed interface.

See calling and usage examples in the `examples\solverf\source` folder of your Intel MKL directory. Specifically, see `ex_nlsqp_f.f` and `ex_nlsqp_bc_f.f`.

Input Parameters

`fcn` User-supplied subroutine to evaluate the function that defines the least squares problem. Call `fcn(m, n, x, f)` with the following parameters:

Parameter	Type	Description
Input Parameters		
<code>m</code>	<code>INTEGER</code>	Length of <code>f</code>
<code>n</code>	<code>INTEGER</code>	Length of <code>x</code>

Parameter	Type	Description
x	REAL for sjacobi DOUBLE PRECISION for djacobi	Array of size n . Vector, at which the function is evaluated. The <code>fcn</code> function should not change this parameter.
Output Parameters		
f	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size m ; contains the function values at x .

You need to declare `fcn` as EXTERNAL in the calling program.

n INTEGER. Length of X .

m INTEGER. Length of F .

x REAL for sjacobi
DOUBLE PRECISION for djacobi
Array of size n . Vector at which the function is evaluated.

eps REAL for sjacobi
DOUBLE PRECISION for djacobi
Precision of the Jacobian matrix calculation.

Output Parameters

$fjac$ REAL for sjacobi
DOUBLE PRECISION for djacobi
Array of size m by n . Contains the Jacobian matrix of the function.

res INTEGER. Indicates task completion status.

- $res = TR_SUCCESS$ - the routine completed the task normally.
- $res = TR_INVALID_OPTION$ - there was an error in the input parameters.
- $res = TR_OUT_OF_MEMORY$ - there was a memory error.

`TR_SUCCESS`, `TR_INVALID_OPTION`, and `TR_OUT_OF_MEMORY` are defined in the `mkl_rci.fi` include file.

See Also

[?jacobix](#)

?jacobix

Alternative interface for ?jacobi function for passing additional data into the objective function.

Syntax

```
res = sjacobix(fcn, n, m, fjac, x, eps, user_data)
```

```
res = djacobix(fcn, n, m, fjac, x, eps, user_data)
```

Include Files

- Fortran: `mkl_rci.fi`, `mkl_rci.f90`

Description

The `?jacobix` routine presents an alternative interface for the `?jacobi` function that enables you to pass additional data into the objective function `fcn`.

See calling and usage examples in the `examples\solverf\source` folder of your Intel MKL directory. Specifically, see `ex_nlsqp_f90_x.f90` and `ex_nlsqp_bc_f90_x.f90`.

Input Parameters

fcn

User-supplied subroutine to evaluate the function that defines the least squares problem. Call `fcn(m, n, x, f, user_data)` with the following parameters:

Parameter	Type	Description
Input Parameters		
<i>m</i>	INTEGER	Length of <i>f</i>
<i>n</i>	INTEGER	Length of <i>x</i>
<i>x</i>	REAL for <code>sjacobix</code> DOUBLE PRECISION for <code>djacobix</code>	Array of size <i>n</i> . Vector, at which the function is evaluated. The <code>fcn</code> function should not change this parameter.
<i>user_data</i>	INTEGER(C_INTPTR_T), for Fortran	Reference to your additional data, if any, passed by value: <code>user_data=%VAL(LOC(data))</code> . Otherwise, a dummy argument.
Output Parameters		
<i>f</i>	REAL for <code>sjacobix</code> DOUBLE PRECISION for <code>djacobix</code>	Array of size <i>m</i> ; contains the function values at <i>x</i> .

You need to declare `fcn` as `EXTERNAL` in the calling program.

n INTEGER. Length of *X*.

m INTEGER. Length of *F*.

x REAL for `sjacobix`
DOUBLE PRECISION for `djacobix`
Array of size *n*. Vector at which the function is evaluated.

eps REAL for `sjacobix`
DOUBLE PRECISION for `djacobix`
Precision of the Jacobian matrix calculation.

user_data INTEGER(C_INTPTR_T). Reference to your additional data, passed by value: *user_data*=%VAL(LOC(*data*)). Otherwise, a dummy argument.

Output Parameters

fjac REAL for sjacobix
DOUBLE PRECISION for djacobix
Array of size m by n). Contains the Jacobian matrix of the function.

res INTEGER. Indicates task completion status.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the `mkl_rci.fi` include file.

See Also

?jacobi

Support Functions

Intel® Math Kernel Library (Intel® MKL) support functions are subdivided into the following groups according to their purpose:

[Version Information](#)

[Threading Control](#)

[Error Handling](#)

[Character Equality Testing](#)

[Timing](#)

[Memory Management](#)

[Single Dynamic Library Control](#)

[Intel® Many Integrated Core \(Intel® MIC\) Architecture Support](#)

[Conditional Numerical Reproducibility Control](#)

[Miscellaneous](#)

The following table lists Intel MKL support functions.

Intel MKL Support Functions

Function Name	Operation
Version Information	
<code>mkl_get_version_string</code>	Returns the Intel MKL version in a character string.
Threading Control	
<code>mkl_set_num_threads</code>	Specifies the number of OpenMP* threads to use.
<code>mkl_domain_set_num_threads</code>	Specifies the number of OpenMP* threads for a particular function domain.
<code>mkl_set_num_threads_local</code>	Specifies the number of OpenMP* threads for all Intel MKL functions on the current execution thread.
<code>mkl_set_dynamic</code>	Enables Intel MKL to dynamically change the number of OpenMP* threads.
<code>mkl_get_max_threads</code>	Gets the number of OpenMP* threads targeted for parallelism.
<code>mkl_domain_get_max_threads</code>	Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.
<code>mkl_get_dynamic</code>	Determines whether Intel MKL is enabled to dynamically change the number of OpenMP* threads.
Error Handling	
<code>xerbla</code>	Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.
<code>pxerbla</code>	Handles error conditions for the ScaLAPACK routines.

Function Name	Operation
<code>mkl_set_exit_handler</code>	Sets the custom handler of fatal errors.
Character Equality Testing	
<code>lsame</code>	Tests two characters for equality regardless of the case.
<code>lsamen</code>	Tests two character strings for equality regardless of the case.
Timing	
<code>second/dsecnd</code>	Returns elapsed time in seconds. Use to estimate real time between two calls to this function.
<code>mkl_get_cpu_clocks</code>	Returns elapsed CPU clocks.
<code>mkl_get_cpu_frequency</code>	Returns CPU frequency value in GHz.
<code>mkl_get_max_cpu_frequency</code>	Returns the maximum CPU frequency value in GHz.
<code>mkl_get_clocks_frequency</code>	Returns the frequency value in GHz based on constant-rate Time Stamp Counter.
Memory Management	
<code>mkl_free_buffers</code>	Frees unused memory allocated by the Intel MKL Memory Allocator.
<code>mkl_thread_free_buffers</code>	Frees unused memory allocated by the Intel MKL Memory Allocator in the current thread.
<code>mkl_mem_stat</code>	Reports the status of the Intel MKL Memory Allocator.
<code>mkl_peak_mem_usage</code>	Reports the peak memory allocated by the Intel MKL Memory Allocator.
<code>mkl_disable_fast_mm</code>	Turns off the Intel MKL Memory Allocator for Intel MKL functions to directly use the system <code>malloc/free</code> functions.
<code>mkl_malloc</code>	Allocates an aligned memory buffer.
<code>mkl_calloc</code>	Allocates and initializes an aligned memory buffer.
<code>mkl_realloc</code>	Changes the size of memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
<code>mkl_free</code>	Frees the aligned memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
<code>mkl_set_memory_limit</code>	On Linux, sets the limit of memory that Intel MKL can allocate for a specified type of memory.
Single Dynamic Library (SDL) Control	
<code>mkl_set_interface_layer</code>	Sets the interface layer for Intel MKL at run time.
<code>mkl_set_threading_layer</code>	Sets the threading layer for Intel MKL at run time.
<code>mkl_set_xerbla</code>	Replaces the error handling routine. Use with SDL on Windows* OS.
<code>mkl_set_progress</code>	Replaces the progress information routine. Use with SDL on Windows* OS.

Function Name	Operation
<code>mkl_set_pardiso_pivot</code>	Replaces the routine handling Intel MKL PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).
Intel MIC Architecture Support	
<code>mkl_mic_enable</code>	Enables Automatic Offload mode.
<code>mkl_mic_disable</code>	Disables Automatic Offload mode.
<code>mkl_mic_get_device_count</code>	Returns the number of Intel® Xeon Phi™ coprocessors on the system when called on the host CPU.
<code>mkl_mic_set_workdivision</code>	For computations in the Automatic Offload mode, sets the fraction of the work for the specified coprocessor or host CPU to do.
<code>mkl_mic_get_workdivision</code>	For computations in the Automatic Offload mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do.
<code>mkl_mic_set_max_memory</code>	Sets the maximum amount of coprocessor memory reserved for Automatic Offload computations.
<code>mkl_mic_free_memory</code>	Frees the coprocessor memory reserved for Automatic Offload computations.
<code>mkl_mic_register_memory</code>	Enables/disables the <code>mkl_malloc</code> function running in Automatic Offload mode to register allocated memory.
<code>mkl_mic_set_device_num_threads</code>	Sets the maximum number of OpenMP* threads to use on an Intel Xeon Phi coprocessor for the Automatic Offload computations.
<code>mkl_mic_set_resource_limit</code>	For computations in the Automatic Offload mode, sets the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.
<code>mkl_mic_get_resource_limit</code>	For computations in the Automatic Offload mode, retrieves the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.
<code>mkl_mic_set_offload_report</code>	Turns on/off reporting of Automatic Offload profiling.
<code>mkl_mic_set_flags</code>	Sets flags to control the behavior of computations in the Automatic Offload mode.
<code>mkl_mic_get_flags</code>	Retrieves flags that control the behavior of computations in the Automatic Offload mode.
<code>mkl_mic_get_status</code>	For the Automatic Offload mode, returns the status of the latest call to an Intel MKL function.
<code>mkl_mic_clear_status</code>	For the Automatic Offload mode, clears the status of the latest call to an Intel MKL function.
<code>mkl_mic_get_meminfo</code>	Retrieves the amount of total and free memory for the specified coprocessor or host CPU.
<code>mkl_mic_get_cpuinfo</code>	Retrieves the number of cores, hardware threads, and frequency for the specified coprocessor or host CPU.
Conditional Numerical Reproducibility (CNR) Control	
<code>mkl_cbwr_set</code>	Configures the CNR mode of Intel MKL.

Function Name	Operation
<code>mkl_cbwr_get</code>	Returns the current CNR settings.
<code>mkl_cbwr_get_auto_branch</code>	Automatically detects the CNR code branch for your platform.
Miscellaneous	
<code>mkl_progress</code>	Tracks computational progress of selective Intel MKL routines.
<code>mkl_enable_instructions</code>	Enables Intel MKL to dispatch a new Intel® architecture on hardware (or simulation) supporting the appropriate instruction set.
<code>mkl_set_env_mode</code>	Sets up the mode that ignores environment settings specific to Intel MKL.
<code>mkl_verbose</code>	Enables or disables Intel MKL Verbose mode.
<code>mkl_set_mpi</code>	Sets the implementation of the message-passing interface to be used by Intel MKL.
<code>mkl_finalize</code>	Terminates Intel MKL execution environment and frees resources allocated by the library.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Using a Fortran Interface Module for Support Functions

To call a support function from your Fortran application, include one of the following statements in your code:

- `INCLUDE mkl_service.fi` or `INCLUDE mkl.fi`
- `USE mkl_service`

The `USE` statement references the `mkl_service.mod` interface module corresponding to your architecture and programming interface. The module provides an application programming interface to Intel MKL support entities, such as subroutines and constants. Because Fortran interface modules are compiler-dependent, Intel MKL offers the `mkl_service.f90` source file for the module, as well as architecture-specific and interface-specific `mkl_service` modules precompiled with the Intel® Fortran or Intel® Visual Fortran compiler. These modules are available in the following subdirectories of the Intel MKL include directory:

Architecture, Interface	Subdirectory of the Intel MKL Installation Directory
IA-32	<code>include\ia32</code>
Intel® 64, LP64	<code>include\intel64\lp64</code>
Intel® 64, ILP64	<code>include\intel64\ilp64</code>

To ensure that your application searches the right module, specify the appropriate subdirectory during compilation as an additional directory for the include path (through the `/I` option on Windows* OS or the `-I` option on Linux* OS or OS X*).

If you are using a non-Intel Fortran compiler, you need to build the module yourself by compiling the `mkl_service.f90` file, available in the Intel MKL include directory.

For more information on compiler-dependent functions and modules, refer to the *Intel MKL User's Guide*.

Version Information

Intel® MKL provides methods for extracting information about the library version number, such as:

- using the `mkl_get_version` function to obtain an `MKLVersion` structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

`mkl_get_version_string`

Returns the Intel MKL version in a character string.

Syntax

```
call mkl_get_version_string( buf )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Output Parameters

Name	Type	Description
<code>buf</code>	<code>CHARACTER*198</code>	Source string

Description

The function returns a string that contains the Intel MKL version.

For usage details, see the code example below:

Example

```
program mkl_get_version_string
character*198  buf

call mkl_get_version_string(buf)
write(*,'(a)') buf

end
```

Threading Control

Intel® MKL provides functions for OpenMP* threading control, discussed in this section.

Important

If Intel MKL operates within the Intel® Threading Building Blocks (Intel® TBB) execution environment, the environment variables for OpenMP* threading control, such as `OMP_NUM_THREADS`, and Intel MKL functions discussed in this section have no effect. If the Intel TBB threading technology is used, control the number of threads through the Intel TBB application programming interface. Read the documentation for the `tbb::task_scheduler_init` class at <https://www.threadingbuildingblocks.org/documentation> to find out how to specify the number of Intel TBB threads.

If Intel® MKL operates within an OpenMP* execution environment, you can control the number of threads for Intel MKL using OpenMP* run-time library routines and environment variables (see the OpenMP* specification for details). Additionally Intel MKL provides *optional* threading control functions and environment variables that enable you to specify the number of threads for Intel MKL and to control dynamic adjustment of the number of threads *independently* of the OpenMP* settings. The settings made with the Intel MKL threading control functions and environment variables do not affect OpenMP* settings but take precedence over them.

If none of the threading control functions is used, Intel MKL environment variables may control Intel MKL threading. For details of those environment variables, see the *Intel MKL User's Guide*.

You can specify the number of threads for Intel MKL function domains with the [mkl_set_num_threads](#) or [mkl_domain_set_num_threads](#) function. While `mkl_set_num_threads` specifies the number of threads for the entire Intel MKL, `mkl_domain_set_num_threads` does it for a specific function domain. The following table lists the function domains that support independent threading control. The table also provides named constants to pass to threading control functions as a parameter that specifies the function domain.

Intel MKL Function Domains

Function Domain	Named Constant
Basic Linear Algebra Subroutines (BLAS)	<code>MKL_DOMAIN_BLAS</code>
Fast Fourier Transform (FFT) functions, except Cluster FFT functions	<code>MKL_DOMAIN_FFT</code>
Vector Math (VM) functions	<code>MKL_DOMAIN_VML</code>
Parallel Direct Solver (PARDISO) functions	<code>MKL_DOMAIN_PARDISO</code>
All Intel MKL functions except the functions from the domains where the number of threads is set explicitly	<code>MKL_DOMAIN_ALL</code>

WARNING

Do not increase the number of OpenMP threads used for Intel MKL PARDISO between the first call to `pardiso` and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.

Both `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions set the number of threads for all subsequent calls to Intel MKL from all applications threads. Use the [mkl_set_num_threads_local](#) function to specify different numbers of threads for Intel MKL on different execution threads of your application. The thread-local settings take precedence over the global settings. However, the thread-local settings may have undesirable side effects (see the description of the `mkl_set_num_threads_local` function for details).

By default, Intel MKL can adjust the specified number of threads dynamically. For example, Intel MKL may use fewer threads if the size of the computation is not big enough or not create parallel regions when running within an OpenMP* parallel region. Although Intel MKL may actually use a different number of threads from the number specified, the library does not create parallel regions with more threads than specified. If dynamic adjustment of the number of threads is disabled, Intel MKL attempts to use the specified number of threads in internal parallel regions (for more information, see the *Intel MKL User's Guide*). Use the [mkl_set_dynamic](#) function to control dynamic adjustment of the number of threads.

mkl_set_num_threads

Specifies the number of OpenMP threads to use.*

Syntax

```
call mkl_set_num_threads( nt )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>nt</code>	INTEGER	$nt > 0$ - The number of threads suggested by the user. $nt \leq 0$ - Invalid value, which is ignored.

Description

This function enables you to specify how many OpenMP threads Intel MKL should use for internal parallel regions. If this number is not set (default), Intel MKL functions use the default number of threads for the OpenMP run-time library. The specified number of threads applies:

- To all Intel MKL functions except the functions from the domains where the number of threads is set with [mkl_domain_set_num_threads](#)
- To all execution threads except the threads where the number of threads is set with [mkl_set_num_threads_local](#)

The number specified is a hint, and Intel MKL may actually use a smaller number.

NOTE

This function takes precedence over the `MKL_NUM_THREADS` environment variable.

Example

```
use mkl_service
...
call mkl_set_num_threads(4)
call my_compute_using_mkl !Intel MKL uses up to 4 OpenMP threads
```

mkl_domain_set_num_threads

Specifies the number of OpenMP threads for a particular function domain.*

Syntax

```
ierr = mkl_domain_set_num_threads( nt, domain )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`

- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>nt</code>	INTEGER	<p><code>nt > 0</code> - The number of threads suggested by the user.</p> <p><code>nt = 0</code> - The default number of threads for the OpenMP run-time library.</p> <p><code>nt < 0</code> - Invalid value, which is ignored.</p>
<code>domain</code>	INTEGER	The named constant that defines the targeted domain.

Description

This function specifies how many OpenMP threads a particular function domain of Intel MKL should use. If this number is not set (default) or if it is set to zero in a call to this function, Intel MKL uses the default number of threads for the OpenMP run-time library. The number of threads specified applies to the specified function domain on all execution threads except the threads where the number of threads is set with `mkl_set_num_threads_local`. For a list of supported values of the `domain` argument, see [Table "Intel MKL Function Domains"](#).

The number of threads specified is only a hint, and Intel MKL may actually use a smaller number.

NOTE

This function takes precedence over the `MKL_DOMAIN_NUM_THREADS` environment variable.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER	<p>1 - Indicates no error, execution is successful.</p> <p>0 - Indicates a failure, possibly because of invalid input parameters.</p>

Example

```

use mkl_service
integer(4) :: status
...
status = mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS)
call my_compute_with_mkl_blas()      !Intel MKL BLAS functions use up to 4 threads
call my_compute_with_mkl_dft()      !Intel MKL FFT functions use the default number of threads

```

`mkl_set_num_threads_local`

Specifies the number of OpenMP threads for all Intel MKL functions on the current execution thread.*

Syntax

```
save_nt = mkl_set_num_threads_local( nt )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`

- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>nt</code>	<code>INTEGER*4</code>	<p><code>nt > 0</code> - The number of threads for Intel MKL functions to use on the current execution thread.</p> <p><code>nt = 0</code> - A request to reset the thread-local number of threads and use the global number.</p>

Description

This function sets the number of OpenMP threads that Intel MKL functions should request for parallel computation. The number of threads is thread-local, which means that it only affects the current execution thread of the application. If the thread-local number is not set or if this number is set to zero in a call to this function, Intel MKL functions use the global number of threads. You can set the global number of threads using the [mkl_set_num_threads](#) or [mkl_domain_set_num_threads](#) function.

The thread-local number of threads takes precedence over the global number: if the thread-local number is non-zero, changes to the global number of threads have no effect on the current thread.

CAUTION

If your application is threaded with OpenMP* and parallelization of Intel MKL is based on nested OpenMP parallelism, different OpenMP parallel regions reuse OpenMP threads. Therefore a thread-local setting in one OpenMP parallel region may continue to affect not only the master thread after the parallel region ends, but also subsequent parallel regions. To avoid performance implications of this side effect, reset the thread-local number of threads before leaving the OpenMP parallel region (see [Examples](#) for how to do it).

Return Values

Name	Type	Description
<code>save_nt</code>	<code>INTEGER*4</code>	The value of the thread-local number of threads that was used before this function call. Zero means that the global number of threads was used.

Examples

This example shows how to avoid the side effect of a thread-local number of threads by reverting to the global setting:

```

use omp_lib
use mkl_service
integer(4) :: dummy
...
call mkl_set_num_threads(16)
call my_compute_using_mkl()      ! Intel MKL functions use up to 16 threads
!$omp parallel num_threads(2)
  if (0 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(4)
  if (1 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(12)
  call my_compute_using_mkl()    ! Intel MKL functions use up to 4 threads on thread 0
                                ! and up to 12 threads on thread 1
!$omp end parallel

```

```
call my_compute_using_mkl()      ! Intel MKL functions use up to 4 threads (!)
dummy = mkl_set_num_threads_local(0)  ! make master thread use global setting
call my_compute_using_mkl()      ! Now Intel MKL functions use up to 16 threads
```

This example shows how to avoid the side effect of a thread-local number of threads by saving and restoring the existing setting:

```
subroutine my_compute(nt)
  use mkl_service
  integer(4) :: nt, save
  save = mkl_set_num_threads_local( nt )    ! save the Intel MKL number of threads
  call my_compute_using_mkl()              ! Intel MKL functions use up to nt threads on this thread
  save = mkl_set_num_threads_local( save )  ! restore the Intel MKL number of threads
end subroutine my_compute
```

mkl_set_dynamic

Enables Intel MKL to dynamically change the number of OpenMP* threads.

Syntax

```
call mkl_set_dynamic( flag )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>flag</i>	INTEGER	<i>flag</i> = 0 - Requests disabling dynamic adjustment of the number of threads. <i>flag</i> ≠ 0 - Requests enabling dynamic adjustment of the number of threads.

Description

This function indicates whether Intel MKL can dynamically change the number of OpenMP threads or should avoid doing this. The setting applies to all Intel MKL functions on all execution threads. This function takes precedence over the `MKL_DYNAMIC` environment variable.

Dynamic adjustment of the number of threads is enabled by default. Specifically, Intel MKL may use fewer threads in parallel regions than the number returned by the `mkl_get_max_threads` function. Disabling dynamic adjustment of the number of threads does not ensure that Intel MKL actually uses the specified number of threads, although the library attempts to use that number.

TIP

If you call Intel MKL from within an OpenMP parallel region and want to create internal parallel regions, either disable dynamic adjustment of the number of threads or set the thread-local number of threads (see `mkl_set_num_threads_local` for how to do it).

Example

```
use mkl_service
...
call mkl_set_num_threads( 8 )
```

```
!$omp parallel
  call my_compute_with_mkl      ! Intel MKL uses 1 thread, being called from OpenMP parallel region
  call mkl_set_dynamic(0)      ! disable adjustment of the number of threads
  call my_compute_with_mkl      ! Intel MKL uses 8 threads
!$omp end parallel
```

mkl_get_max_threads

Gets the number of OpenMP threads targeted for parallelism.*

Syntax

```
nt = mkl_get_max_threads()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function returns the number of OpenMP threads for Intel MKL to use in internal parallel regions. This number depends on whether dynamic adjustment of the number of threads by Intel MKL is disabled (by an environment setting or in a function call):

- If the dynamic adjustment is disabled, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:
 - A call to [mkl_set_num_threads_local](#)
 - The last of the calls to [mkl_set_num_threads](#) or [mkl_domain_set_num_threads\(..., MKL_DOMAIN_ALL\)](#)
 - The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
 - The `MKL_NUM_THREADS` environment variable
 - A call to `omp_set_num_threads`
 - The `OMP_NUM_THREADS` environment variable
- If the dynamic adjustment is enabled, the function returns the number of physical cores on your system.

The number of threads returned by this function is a hint, and Intel MKL may actually use a different number.

Return Values

Name	Type	Description
<code>nt</code>	<code>INTEGER*4</code>	The maximum number of threads for Intel MKL functions to use in internal parallel regions.

Example

```
use mkl_service
...
if (1 == mkl_get_max_threads()) print *, "Intel MKL does not employ threading"
```

See Also

[mkl_set_dynamic](#)

[mkl_get_dynamic](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_domain_get_max_threads

Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.

Syntax

```
nt = mkl_domain_get_max_threads( domain )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>domain</i>	INTEGER	The named constant that defines the targeted domain.

Description

Computational functions of the Intel MKL function domain defined by the *domain* parameter use the value returned by this function as a limit of the number of OpenMP threads they should request for parallel computations. The `mkl_domain_get_max_threads` function returns the thread-local number of threads or, if that value is zero or not set, the global number of threads. To determine this number, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:

- A call to `mkl_set_num_threads_local`
- The last of the calls to `mkl_set_num_threads` or `mkl_domain_set_num_threads(..., MKL_DOMAIN_ALL)`
- A call to `mkl_domain_set_num_threads(..., domain)`
- The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
- The `MKL_DOMAIN_NUM_THREADS` environment variable (with the specific domain tag)
- The `MKL_NUM_THREADS` environment variable
- A call to `omp_set_num_threads`
- The `OMP_NUM_THREADS` environment variable

Actual number of threads used by the Intel MKL computational functions may vary depending on the problem size and on whether dynamic adjustment of the number of threads is enabled (see the description of `mkl_set_dynamic`). For a list of supported values of the *domain* argument, see [Table "Intel MKL Function Domains"](#).

Return Values

Name	Type	Description
<i>nt</i>	INTEGER*4	The maximum number of threads for Intel MKL functions from a given domain to use in internal parallel regions. If an invalid value of <i>domain</i> is supplied, the function returns the number of threads for <code>MKL_DOMAIN_ALL</code>

Example

```

use mkl_service
...
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS)) then
print *, "Intel MKL BLAS functions employ threading"
end if

```

mkl_get_dynamic

Determines whether Intel MKL is enabled to dynamically change the number of OpenMP threads.*

Syntax

```
ret = mkl_get_dynamic()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function returns the status of dynamic adjustment of the number of OpenMP* threads. To determine this status, the function inspects the return value of the following function call and if it is undefined, inspects the environment setting below:

- A call to [mkl_set_dynamic](#)
- The `MKL_DYNAMIC` environment variable

NOTE

Dynamic adjustment of the number of threads is enabled by default.

The dynamic adjustment works as follows. Suppose that the [mkl_get_max_threads](#) function returns the number of threads equal to N . If dynamic adjustment is enabled, Intel MKL may request up to N threads, depending on the size of the problem. If dynamic adjustment is disabled, Intel MKL requests exactly N threads for internal parallel regions (provided it uses a threaded algorithm with at least N computations that can be done in parallel). However, the OpenMP* run-time library may be configured to supply fewer threads than Intel MKL requests, depending on the OpenMP* setting of dynamic adjustment.

Return Values

Name	Type	Description
<code>ret</code>	INTEGER*4	0 - Dynamic adjustment of the number of threads is disabled. 1 - Dynamic adjustment of the number of threads is enabled.

Example

```

use mkl_service
integer(4) :: nt
...
nt = mkl_get_max_threads()
if (1 == mkl_get_dynamic()) then

```

```

print ('"Intel MKL may use less than "I0" threads for a large problem"', nt
else
print ('"Intel MKL should use "I0" threads for a large problem"', nt
end if

```

Error Handling

Error Handling for Linear Algebra Routines

xerbla

Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

Syntax

call xerbla(*sname*, *info*)

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
<i>sname</i>	CHARACTER* (*)	The name of the routine that called xerbla
<i>info</i>	INTEGER	The position of the invalid parameter in the parameter list of the calling function or an error code

Description

The xerbla function is an error handler for Intel MKL BLAS, LAPACK, Vector Math, and Vector Statistics functions. These functions call xerbla if an issue is encountered on entry or during the function execution.

xerbla operates as follows:

1. Prints a message that depends on the value of the *info* parameter as explained in the [Error Messages Printed by xerbla](#) table.

NOTE

A specific message can differ from the listed messages in numeric values and/or function names.

2. Returns to the calling application.

Comments in the Netlib LAPACK reference code (http://www.netlib.org/lapack/explore-html/d1/dc0/_b_l_a_s_2_s_r_c_2xerbla_8f.html) suggest this behavior although the LAPACK User's Guide recommends that the execution should stop when an error occurs.

Error Messages Printed by xerbla

Value of <i>info</i>	Error Message
1001	Intel MKL ERROR: Incompatible optional parameters on entry to DGEMM.

Value of <i>info</i>	Error Message
1212	Intel MKL INTERNAL ERROR: Issue accessing coprocessor in function CGEEV.
1000 or 1089	Intel MKL INTERNAL ERROR: Insufficient workspace available in function CGELSD.
< 0	Intel MKL INTERNAL ERROR: Condition 1 detected in function DLASD8.
Other	Intel MKL ERROR: Parameter 6 was incorrect on entry to DGEMM.

Note that `xerbla` is an internal function. You can change or disable printing of an error message by providing your own `xerbla` function. The following examples illustrate usage of `xerbla`.

Example

```
subroutine xerbla (sname, info)
character*(*) sname !Name of subprogram that called xerbla
integer*4 info !Position of the invalid parameter in the parameter list
return !Return to the calling subprogram end
end
```

See Also

[mkl_set_xerbla](#)

pxerbla

Error handling routine called by ScaLAPACK routines.

Syntax

```
void pxerbla(MKL_INT* ictxt, char* sname, MKL_INT* info);
```

Include Files

Input Parameters

<i>ictxt</i>	(global) MKL_INT The BLACS context handle, indicating the global context of the operation. The context itself is global.
<i>sname</i>	(global) char The name of the routine which called <code>pxerbla</code> .
<i>info</i>	(global) MKL_INT. The position of the invalid parameter in the parameter list of the calling routine.

Description

This routine is an error handler for the *ScaLAPACK* routines. It is called if an input parameter has an invalid value. A message is printed and program execution continues. For *ScaLAPACK* driver and computational routines, a `RETURN` statement is issued following the call to `pxerbla`.

Control returns to the higher-level calling routine, and you can determine how the program should proceed. However, in the specialized low-level *ScaLAPACK* routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pxerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a non-zero value of *info* on return from a ScaLAPACK routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Handling Fatal Errors

A fatal error is a circumstance under which Intel MKL cannot continue the computation. For example, a fatal error occurs when Intel MKL cannot load a dynamic library or confronts an unsupported CPU type. In case of a fatal error, the default Intel MKL behavior is to print an explanatory message to the console and call an internal function that terminates the application with a call to the system `exit()` function. Intel MKL enables you to override this behavior by setting a custom handler of fatal errors. The custom error handler can be configured to throw a C++ exception, set a global variable indicating the failure, or otherwise handle cannot-continue situations. It is not necessary for the custom error handler to call the system `exit()` function. Once execution of the error handler completes, a call to Intel MKL returns to the calling program without performing any computations and leaves no memory allocated by Intel MKL and no thread synchronization pending on return.

To specify a custom fatal error handler, call the `mkl_set_exit_handler` function.

`mkl_set_exit_handler`

Sets the custom handler of fatal errors.

Syntax

```
external :: myexit
interface = mkl_set_exit_handler( myexit )
```

Fortran Include Files/Modules

None.

Input Parameters

Name	Interface	Description
<code>myexit</code>	<pre>interface subroutine myexit(iwhy) integer,value :: iwhy end subroutine myexit end interface</pre>	The error handler to set.

Description

This function sets the custom handler of fatal errors.

The following example shows how to use a custom handler of fatal errors in your application:

```
subroutine myexit(rsn)
integer,value :: rsn
call msgbox("Application is terminating")
end myexit

program app
external :: myexit
call mkl_set_exit_handler(myexit)
!... compute using Intel MKL...
```

Character Equality Testing

lsame

Tests two characters for equality regardless of the case.

Syntax

```
val = lsame( ca, cb )
```

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
<i>ca, cb</i>	CHARACTER*1	The single characters to be compared

Description

This logical function checks whether two characters are equal regardless of the case.

Return Values

Name	Type	Description
<i>val</i>	LOGICAL	Result of the comparison: <ul style="list-style-type: none"> • .TRUE. if <i>ca</i> is the same letter as <i>cb</i>, maybe except for the case. • .FALSE. if <i>ca</i> and <i>cb</i> are different letters for whatever cases.

lsamen

Tests two character strings for equality regardless of the case.

Syntax

```
val = lsamen( n, ca, cb )
```

Include Files

- mkl.fi

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER	The number of characters in <i>ca</i> and <i>cb</i> to be compared.
<i>ca, cb</i>	CHARACTER* (*)	Character strings of length at least <i>n</i> to be compared. Only the first <i>n</i> characters of each string will be accessed.

Description

This logical function tests whether the first n letters of one string are the same as the first n letters of the other string, regardless of the case.

Return Values

Name	Type	Description
<code>val</code>	LOGICAL	Result of the comparison: <ul style="list-style-type: none"> • <code>.TRUE.</code> if the first n letters in <code>ca</code> and <code>cb</code> character strings are equal, maybe except for the case, or if the length of character string <code>ca</code> or <code>cb</code> is less than n. • <code>.FALSE.</code> if the first n letters in <code>ca</code> and <code>cb</code> character strings are different for whatever cases.

Timing

`second/dsecnd`

Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

Syntax

`val = second()`

`val = dsecnd()`

Include Files

- `mkl.fi`

Description

The `second/dsecnd` function returns time in seconds to be used to estimate real time between two calls to the function. The difference between these functions is in the precision of the floating-point type of the result: while `second` returns the single-precision type, `dsecnd` returns the double-precision type.

Use these functions to measure durations. To do this, call each of these functions twice. For example, to measure performance of a routine, call the appropriate function directly before a call to the routine to be measured, and then after the call of the routine. The difference between the returned values shows real time spent in the routine.

Initializations may take some time when the `second/dsecnd` function runs for the first time. To eliminate the effect of this extra time on your measurements, make the first call to `second/dsecnd` in advance.

Do not use `second` to measure short time intervals because the single-precision format is not capable of holding sufficient timer precision.

Return Values

Name	Type	Description
<code>val</code>	REAL for <code>second</code> DOUBLE PRECISION for <code>dsecnd</code>	Elapsed real time in seconds

mkl_get_cpu_clocks

Returns elapsed CPU clocks.

Syntax

```
call mkl_get_cpu_clocks( clocks )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Output Parameters

Name	Type	Description
<code>clocks</code>	INTEGER*8	Elapsed CPU clocks

Description

The `mkl_get_cpu_clocks` function returns the elapsed CPU clocks.

This may be useful when timing short intervals with high resolution. The `mkl_get_cpu_clocks` function is also applied in pairs like `second/dsecond`. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_get_cpu_frequency

Returns the current CPU frequency value in GHz.

Syntax

```
freq = mkl_get_cpu_frequency()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The function `mkl_get_cpu_frequency` returns the current CPU frequency in GHz.

NOTE

The returned value may vary from run to run if power management or Intel® Turbo Boost Technology is enabled.

Return Values

Name	Type	Description
<i>freq</i>	DOUBLE PRECISION	Current CPU frequency value in GHz

mkl_get_max_cpu_frequency

Returns the maximum CPU frequency value in GHz.

Syntax

```
freq = mkl_get_max_cpu_frequency()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The function `mkl_get_max_cpu_frequency` returns the maximum CPU frequency in GHz.

Return Values

Name	Type	Description
<i>freq</i>	DOUBLE PRECISION	Maximum CPU frequency value in GHz

mkl_get_clocks_frequency

Returns the frequency value in GHz based on constant-rate Time Stamp Counter.

Syntax

```
freq = mkl_get_clocks_frequency()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The function `mkl_get_clocks_frequency` returns the CPU frequency value (in GHz) based on constant-rate Time Stamp Counter (TSC). Use of the constant-rate TSC ensures that each clock tick is constant even if the CPU frequency changes. Therefore, the returned frequency is constant.

NOTE

Obtaining the frequency may take some time when `mkl_get_clocks_frequency` is called for the first time. The same holds for functions `second/dsecnd`, which call `mkl_get_clocks_frequency`.

Return Values

Name	Type	Description
<code>freq</code>	DOUBLE PRECISION	Frequency value in GHz

See Also

[second/dsecnd](#)

[Using a Fortran Interface Module for Support Functions](#)

Memory Management

This section describes the Intel MKL memory functions. See the *Intel® MKL User's Guide* for more memory usage information.

mkl_free_buffers

Frees unused memory allocated by the Intel MKL Memory Allocator.

Syntax

```
call mkl_free_buffers
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

To improve performance of Intel MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_free_buffers` function frees unused memory allocated by the Memory Allocator.

See the *Intel MKL User's Guide* for details.

You should call `mkl_free_buffers` after the last call to Intel MKL functions. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel MKL functions.

In a threaded application, avoid calling `mkl_free_buffers` from each thread because the function has a global effect. Call `mkl_thread_free_buffers` instead.

Usage of mkl_free_buffers with FFT Functions (C Example)

```
DFTI_DESCRIPTOR_HANDLE hand1;
DFTI_DESCRIPTOR_HANDLE hand2;
void mkl_free_buffers(void);
. . . . .
/* Using MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
```

```
Status = DftiComputeForward(hand1, s_array1);
. . . . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . . . .
Status = DftiFreeDescriptor(&hand1);
/* Do not call mkl_free_buffers() here because the hand2 descriptor will be corrupted! */
. . . . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here you finish using Intel MKL FFT */
/* Memory leak will be triggered by any memory control tool */
/* Use mkl_free_buffers() to avoid memory leaking */
mkl_free_buffers();
```

mkl_thread_free_buffers

Frees unused memory allocated by the Intel MKL Memory Allocator in the current thread.

Syntax

call mkl_thread_free_buffers

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Description

To improve performance of Intel MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_thread_free_buffers` function frees unused memory allocated by the Memory Allocator in the current thread only.

You should call `mkl_thread_free_buffers` after the last call to Intel MKL functions in the current thread. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel MKL functions.

See Also

[mkl_free_buffers](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_disable_fast_mm

Turns off the Intel MKL Memory Allocator for Intel MKL functions to directly use the system malloc/free functions.

Syntax

`mm = mkl_disable_fast_mm`

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod

- Module (source): `mkl_service.f90`

Description

The `mkl_disable_fast_mm` function turns the Intel MKL Memory Allocator off for Intel MKL functions to directly use the system `malloc/free` functions. Intel MKL Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The Memory Allocator is turned on by default for better performance. To turn it off, you can use the `mkl_disable_fast_mm` function or the `MKL_DISABLE_FAST_MM` environment variable (See the *Intel MKL User's Guide* for details.) Call `mkl_disable_fast_mm` before calling any Intel MKL functions that require allocation of memory buffers.

NOTE

Turning the Memory Allocator off negatively impacts performance of some Intel MKL routines, especially for small problem sizes.

Return Values

Name	Type	Description
<code>mm</code>	INTEGER*4	1 - The Memory Allocator is successfully turned off. 0 - Turning the Memory Allocator off failed.

`mkl_mem_stat`

Reports the status of the Intel MKL Memory Allocator.

Syntax

```
AllocatedBytes = mkl_mem_stat( AllocatedBuffers )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Output Parameters

Name	Type	Description
<code>AllocatedBuffers</code>	INTEGER*4	The number of buffers allocated by Intel MKL.

Description

The function returns the number of buffers allocated by Intel MKL and the amount of memory in these buffers. Intel MKL can allocate the memory buffers internally or in a call to `mkl_malloc/mkl_calloc`. If no buffers are allocated at the moment, the `mkl_mem_stat` function returns 0. Call `mkl_mem_stat` to check the Intel MKL memory status.

NOTE

If you free all the memory allocated in calls to `mkl_malloc` or `mkl_calloc` and then call `mkl_free_buffers`, a subsequent call to `mkl_mem_stat` normally returns 0.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	INTEGER*8	The amount of allocated memory (in bytes).

See Also

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_peak_mem_usage

Reports the peak memory allocated by the Intel MKL Memory Allocator.

Syntax

```
AllocatedBytes = mkl_peak_mem_usage( mode )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>mode</i>	INTEGER*4	Requested mode of the function's operation. Possible values: <ul style="list-style-type: none"> • <code>MKL_PEAK_MEM_ENABLE</code> - start gathering the peak memory data • <code>MKL_PEAK_MEM_DISABLE</code> - stop gathering the peak memory data • <code>MKL_PEAK_MEM</code> - return the peak memory • <code>MKL_PEAK_MEM_RESET</code> - return the peak memory and reset the counter to start gathering the peak memory data from scratch

Description

The `mkl_peak_mem_usage` function reports the peak memory allocated by the Intel MKL Memory Allocator.

Gathering the peak memory data is turned off by default. If you need to know the peak memory, explicitly turn the data gathering mode on by calling the function with the `MKL_PEAK_MEM_ENABLE` value of the parameter. Use the `MKL_PEAK_MEM` and `MKL_PEAK_MEM_RESET` values only when the data gathering mode is turned on. Otherwise the function returns -1. The data gathering mode leads to performance degradation, so when the mode is turned on, you can turn it off by calling the function with the `MKL_PEAK_MEM_DISABLE` value of the parameter.

- If Intel MKL is running in a threaded mode, the `mkl_peak_mem_usage` function may return different amounts of memory from run to run.
- The function reports the peak memory for the entire application, not just for the calling thread.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	INTEGER*8	The peak memory allocated by the Memory Allocator (in bytes) or -1 in case of errors.

See Also

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_malloc

Allocates an aligned memory buffer.

Syntax

```
a_ptr = mkl_malloc( alloc_size, alignment )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>alloc_size</i>	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	Size of the buffer to be allocated.
<i>alignment</i>	INTEGER*4	Alignment of the buffer.

Description

The function allocates an *alloc_size*-byte buffer aligned on the *alignment*-byte boundary.

If *alignment* is not a power of 2, the 32-byte alignment is used.

Return Values

Name	Type	Description
<i>a_ptr</i>	POINTER	Pointer to the allocated buffer if <i>alloc_size</i> ≥ 1, NULL if <i>alloc_size</i> < 1.

See Also

[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_calloc

Allocates and initializes an aligned memory buffer.

Syntax

```
a_ptr = mkl_calloc( num, size, alignment )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>num</i>	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	The number of elements in the buffer to be allocated.
<i>size</i>	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	The size of the element.
<i>alignment</i>	INTEGER*4	Alignment of the buffer.

Description

The function allocates a $num * size$ -byte buffer, aligned on the *alignment*-byte boundary, and initializes the buffer with zeros.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
<i>a_ptr</i>	POINTER	Pointer to the allocated buffer if $size \geq 1$, NULL if $size < 1$.

See Also

[mkl_malloc](#)

[mkl_realloc](#)

[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_realloc

Changes the size of memory buffer allocated by mkl_malloc/mkl_calloc.

Syntax

```
a_ptr = mkl_realloc( ptr, size )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>ptr</i>	POINTER	Pointer to the memory buffer allocated by the <code>mkl_malloc</code> or <code>mkl_calloc</code> function or a NULL pointer.
<i>size</i>	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems	New size of the buffer.

Description

The function changes the size of the memory buffer allocated by the `mkl_malloc` or `mkl_calloc` function to *size* bytes. The first bytes of the returned buffer up to the minimum of the old and new sizes keep the content of the input buffer. The returned memory buffer can have a different location than the input one. If *ptr* is NULL, the function works as `mkl_malloc`.

Return Values

Name	Type	Description
<i>a_ptr</i>	POINTER	Pointer to the re-allocated buffer.

See Also

[mkl_malloc](#)

[mkl_calloc](#)

[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_free

Frees the aligned memory buffer allocated by `mkl_malloc/mkl_calloc`.

Syntax

```
call mkl_free( a_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>a_ptr</i>	POINTER	Pointer to the buffer to be freed.

Description

The function frees the buffer pointed by *a_ptr* and allocated by the `mkl_malloc()` or `mkl_calloc()` function and does nothing if *a_ptr* is NULL.

See Also

[mkl_malloc](#)

[mkl_calloc](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_set_memory_limit

On Linux, sets the limit of memory that Intel MKL can allocate for a specified type of memory.

Syntax

```
stat = mkl_set_memory_limit( mem_type, limit )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>mem_type</i>	INTEGER*4	Type of memory to limit. Possible values: MKL_MEM_MCDRAM - Multi-Channel Dynamic Random Access Memory (MCDRAM) (for more details, see http://www.anandtech.com/show/8217/intels-knights-landing-coprocessor-detailed).
<i>limit</i>	INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems.	Memory limit in megabytes.

Description

This function sets the limit for the amount of memory that Intel MKL can allocate for the specified memory type. The limit bounds both internal allocations (inside Intel MKL computation routines) and external allocations (in a call to `mkl_malloc`, `mkl_calloc`, or `mkl_realloc`). By default no limit is set for memory allocation.

Call `mkl_set_memory_limit` at most once, prior to calling any other Intel MKL function in your application except `mkl_set_interface_layer` and `mkl_set_threading_layer`.

-
- Allocation in MCDRAM requires `libmemkind` and `libjemalloc` dynamic libraries which are a part of Intel® Manycore Platform Software Package (Intel® MPSP) for Linux*.
 - The `mkl_set_memory_limit` function takes precedence over the `MKL_FAST_MEMORY_LIMIT` environment variable.
-

Return Values

Type	Description
INTEGER*4	Status of the function completion: <ul style="list-style-type: none"> • 1 - the limit is set • 0 - the limit is not set

See Also

[mkl_malloc](#)

[mkl_calloc](#)

[mkl_realloc](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

Usage Examples for the Memory Functions

Usage Example for 1-dimensional Arrays

```

PROGRAM FOO
  INCLUDE 'mkl.fi'

  DOUBLE PRECISION    A,B,C
  POINTER      (A_PTR,A(1)), (B_PTR,B(1)), (C_PTR,C(1))
  INTEGER      N, I
  REAL*8      ALPHA, BETA
  INTEGER*8    ALLOCATED_BYTES
  INTEGER*8    PEAK_MEMORY
  INTEGER*4    ALLOCATED_BUFFERS

#ifdef _SYSTEM_BITS32
  INTEGER*4    MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
  INTEGER*4    ALLOC_SIZE, NUM, SIZE
#else
  INTEGER*8    MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
  INTEGER*8    ALLOC_SIZE, NUM, SIZE
#endif

  EXTERNAL    MKL_MALLOC, MKL_FREE, MKL_CALLOC, MKL_REALLOC

  ALPHA = 1.1; BETA = -1.2
  N = 1000
  SIZE = 8
  NUM = N*N
  ALLOC_SIZE = SIZE*NUM
  PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_ENABLE)
  A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
  B_PTR = MKL_MALLOC(ALLOC_SIZE,64)
  C_PTR = MKL_CALLOC(NUM,SIZE,64)
  DO I=1,N*N
    A(I) = I
    B(I) = -I
  END DO

  CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

  ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
  PRINT *, 'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',

```

```

$ ALLOCATED_BUFFERS, ' buffers '

CALL MKL_FREE_BUFFERS
CALL MKL_FREE(A_PTR)
CALL MKL_FREE(B_PTR)
CALL MKL_FREE(C_PTR)

ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
IF (ALLOCATED_BYTES > 0) THEN
  PRINT *, 'MKL MEMORY LEAK!'
  PRINT *, 'AFTER MKL_FREE_BUFFERS there are ',
$   ALLOCATED_BYTES, ' bytes in ',
$   ALLOCATED_BUFFERS, ' buffers'
END IF

PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_RESET)
PRINT *, 'Peak memory allocated by Intel MKL memory allocator ',
$ PEAK_MEMORY, ' bytes. ',
$ 'Start to count new memory peak'

A_PTR = MKL_MALLOC(ALLOC_SIZE, 64)
A_PTR = MKL_REALLOC(A_PTR, ALLOC_SIZE*SIZE)
CALL MKL_FREE(A_PTR)
PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM)
PRINT *, 'After reset of peak memory counter',
$ 'Peak memory allocated by Intel MKL memory allocator ',
$ PEAK_MEMORY, ' bytes'

STOP
END

```

Usage Example for 2-dimensional Arrays

```

PROGRAM FOO
INTEGER    N
PARAMETER (N=100)
DOUBLE PRECISION  A,B,C
POINTER     (A_PTR,A(N,*)), (B_PTR,B(N,*)), (C_PTR,C(N,*))
INTEGER     I,J
REAL*8      ALPHA, BETA
INTEGER*8    ALLOCATED_BYTES
INTEGER*4    ALLOCATED_BUFFERS

#ifdef _SYSTEM_BITS32
  INTEGER*4 MKL_MALLOC
  INTEGER*4 ALLOC_SIZE
#else
  INTEGER*8 MKL_MALLOC
  INTEGER*8 ALLOC_SIZE
#endif

INTEGER     MKL_MEM_STAT
EXTERNAL    MKL_MALLOC, MKL_FREE, MKL_MEM_STAT

ALPHA = 1.1; BETA = -1.2
ALLOC_SIZE = 8*N*N
A_PTR = MKL_MALLOC(ALLOC_SIZE, 64)
B_PTR = MKL_MALLOC(ALLOC_SIZE, 64)
C_PTR = MKL_MALLOC(ALLOC_SIZE, 64)
DO I=1,N
  DO J=1,N

```

```

      A(I,J) = I
      B(I,J) = -I
      C(I,J) = 0.0
    END DO
  END DO

  CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

  ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
  PRINT *, 'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',
$  ALLOCATED_BUFFERS,' buffers '

  CALL MKL_FREE_BUFFERS
  CALL MKL_FREE(A_PTR)
  CALL MKL_FREE(B_PTR)
  CALL MKL_FREE(C_PTR)

  ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
  IF (ALLOCATED_BYTES > 0) THEN
    PRINT *, 'MKL MEMORY LEAK!'
    PRINT *, 'AFTER MKL_FREE_BUFFERS there are ',
$  ALLOCATED_BYTES,' bytes in ',
$  ALLOCATED_BUFFERS,' buffers'
  END IF

  STOP
  END

```

Single Dynamic Library Control

Intel® MKL provides the Single Dynamic Library (SDL), which enables setting the interface and threading layer for Intel MKL at run time. See *Intel® MKL User's Guide* for details of SDL and layered model concept. This section describes the functions supporting SDL.

mkl_set_interface_layer

Sets the interface layer for Intel MKL at run time. Use with the Single Dynamic Library.

Syntax

```
interface = mkl_set_interface_layer( required_interface )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>required_interface</code>	INTEGER	Determines the interface layer. Possible values depend on the system architecture. Some of the values are only available on Linux* OS: <ul style="list-style-type: none"> • Intel® 64 architecture:

Name	Type	Description
		MKL_INTERFACE_LP64 for the Intel LP64 interface.
		MKL_INTERFACE_ILP64 for the Intel ILP64 interface.
		MKL_INTERFACE_LP64+MKL_INTERFACE_GNU for the GNU* LP64 interface on Linux OS.
		MKL_INTERFACE_ILP64+MKL_INTERFACE_GNU for the GNU ILP64 interface on Linux OS.
		<ul style="list-style-type: none"> IA-32 architecture: <ul style="list-style-type: none"> MKL_INTERFACE_LP64 for the Intel interface on Linux OS. MKL_INTERFACE_LP64+MKL_INTERFACE_GNU or MKL_INTERFACE_GNU for the GNU interface on Linux OS.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_interface_layer` function sets the specified interface layer for Intel MKL at run time.

Call this function prior to calling any other Intel MKL function in your application except `mkl_set_threading_layer`. You can call `mkl_set_interface_layer` and `mkl_set_threading_layer` in any order.

The `mkl_set_interface_layer` function takes precedence over the `MKL_INTERFACE_LAYER` environment variable.

See *Intel MKL User's Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
INTEGER	<ul style="list-style-type: none"> Current interface layer if it is set in a call to <code>mkl_set_interface_layer</code> or specified by environment variables or defaults. <p>Possible values are specified in Input Parameters.</p> -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_threading_layer`

Sets the threading layer for Intel MKL at run time. Use with the Single Dynamic Library (SDL).

Syntax

```
threading = mkl_set_threading_layer( required_threading )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>required_threading</code>	INTEGER	<p>Determines the threading layer. Possible values:</p> <p><code>MKL_THREADING_INTEL</code> for Intel threading.</p> <p><code>MKL_THREADING_SEQUENTIAL</code> for the sequential mode of Intel MKL.</p> <p><code>MKL_THREADING_TBB</code> for threading with the Intel® Threading Building Blocks.</p> <p><code>MKL_THREADING_PGI</code> for PGI threading on Windows* or Linux* operating system only. Do not use this value with the SDL for Intel® Many Integrated Core (Intel® MIC) Architecture.</p> <p><code>MKL_THREADING_GNU</code> for GNU threading on Linux* operating system only. Do not use this value with the SDL for Intel MIC Architecture.</p>

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_threading_layer` function sets the specified threading layer for Intel MKL at run time.

Call this function prior to calling any other Intel MKL function in your application except `mkl_set_interface_layer`.

You can call `mkl_set_threading_layer` and `mkl_set_interface_layer` in any order.

The `mkl_set_threading_layer` function takes precedence over the `MKL_THREADING_LAYER` environment variable.

See *Intel MKL User's Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
INTEGER	<ul style="list-style-type: none"> Current threading layer if it is set in a call to <code>mkl_set_threading_layer</code> or specified by environment variables or defaults. Possible values are specified in Input Parameters. -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_xerbla`

Replaces the error handling routine. Use with the Single Dynamic Library on Windows OS.*

Syntax

```
old_xerbla_ptr = mkl_set_xerbla( new_xerbla_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>new_xerbla_ptr</code>	XerblaEntry	Pointer to the error handling routine to be used.

Description

If you are linking with the Single Dynamic Library (SDL) `mkl_rt.lib` on Windows* OS, the `mkl_set_xerbla` function replaces the error handling routine that is called by Intel MKL functions with the routine specified by the parameter.

See *Intel MKL User's Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced error handling routine.

See Also

[xerbla](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_set_progress

Replaces the progress information routine. Use with the Single Dynamic Library (SDL) on Windows OS.*

Syntax

```
old_progress_ptr mkl_set_progress( new_progress_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>new_progress_ptr</code>	ProgressEntry	Pointer to the progress information routine to be used.

Description

If you are linking with the Single Dynamic Library (SDL) `mkl_rt.lib` on Windows* OS, the `mkl_set_progress` function replaces the currently used progress information routine with the routine specified by the parameter.

See *Intel MKL User's Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced progress information routine.

See Also

[mkl_progress](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_set_pardiso_pivot

Replaces the routine handling Intel MKL PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).

Syntax

```
old_pardiso_pivot_ptr = mkl_set_pardiso_pivot( new_pardiso_pivot_ptr )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>new_pardiso_pivot_ptr</code>	<code>PardisoPivotEntry</code>	Pointer to the pivot setting routine to be used.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_pardiso_pivot` function replaces the pivot setting routine that is called by Intel MKL functions with the routine specified by the parameter.

See *Intel MKL User's Guide* for usage details of the SDL.

Return Values

Type	Description
<code>PardisoPivotEntry</code>	Pointer to the replaced pivot setting routine.

See Also

[mkl_pardiso_pivot](#)

Intel Many Integrated Core Architecture Support

This section describes Intel MKL functions to support the use of Intel Xeon Phi coprocessors and especially Automatic Offload mode, when the computations are automatically offloaded to the coprocessors.

The `mkl_mic_enable` function enables Intel MKL to offload computations to Intel Xeon Phi coprocessors automatically, while the `mkl_mic_disable` function disables automatic offloading.

Important

Automatic Offload supports only OpenMP* threaded Intel MKL.

Optional work-division control functions enable you to specify the fractional amount of work to distribute between the host CPU and the coprocessors in the Automatic Offload mode. Work division is a fractional measure ranging from 0.0 to 1.0. For example, setting work division for the host CPU to 0.5 means to keep half of the computational work on the host CPU and move half to the coprocessor(s). Setting work division to 0.25 for a coprocessor means to offload a quarter of the computational work to this coprocessor while leaving the rest on the host CPU.

Other functions enable more control over computational resources of Intel Xeon Phi coprocessors.

NOTE

The support functions for Intel MIC Architecture take precedence over the respective environment variables.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_mic_enable

Enables Automatic Offload mode.

Syntax

```
ierr = mkl_mic_enable( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The `mkl_mic_enable` function enables Automatic Offload mode and initializes Intel Xeon Phi coprocessors available on your system. This function takes precedence over the `MKL_MIC_ENABLE` environment variable. Unlike the function, the environment variable enables Automatic Offload mode, but the initialization is delayed until a call to an Intel MKL function supporting Automatic Offload. For functions supporting Automatic Offload, see the *Intel MKL Release Notes*.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	Result status: = 0 Indicates that Automatic Offload mode is successfully enabled. < 0 Indicates a failure to enable Automatic Offload mode.

mkl_mic_disable

Disables Automatic Offload mode.

Syntax

```
ierr = mkl_mic_disable( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The `mkl_mic_disable` function disables Automatic Offload mode. To enable Automatic Offload mode back, use the `mkl_mic_enable` function.

NOTE

If Automatic Offload mode is not enabled, a call to `mkl_mic_disable` completes successfully if Intel Xeon Phi coprocessors are available on your system.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	Result status: = 0 Indicates that Automatic Offload mode is successfully disabled. < 0 Indicates a failure to disable Automatic Offload mode.

`mkl_mic_get_device_count`

Returns the number of Intel Xeon Phi coprocessors on the system when called on the host CPU.

Syntax

```
ndevices = mkl_mic_get_device_count( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The `mkl_get_device_count` function returns the number of Intel Xeon Phi coprocessors on your system. You may need this number to specify a custom work-division scheme.

CAUTION

Call this function only on the host CPU.

Return Values

Name	Type	Description
<code>ndevices</code>	INTEGER*4	The number of Intel Xeon Phi coprocessors available on the system. Equals zero if there are none.

See Also

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_set_workdivision

For computations in the Automatic Offload mode, sets the fraction of the work for the specified coprocessor or host CPU to do.

Syntax

```
ierr = mkl_mic_set_workdivision( target_type, target_number, wd )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use one of the following values: <ul style="list-style-type: none"> • <code>MKL_TARGET_HOST</code> - host CPU • <code>MKL_TARGET_MIC</code> - Intel Xeon Phi coprocessor, default
<code>target_number</code>	INTEGER*4	The device to set the fraction of work for. Takes the following values: <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2. • < 0. Reserved. <p>If <code>target_type = MKL_TARGET_HOST</code>, the function ignores the <code>target_number</code> parameter, which may have any value.</p>
<code>wd</code>	REAL*8	The fractional amount of the work that the specified device should do, where $0.0 \leq wd \leq 1.0$. Specifying <code>MKL_MIC_AUTO_WORKDIVISION</code> for <code>wd</code> indicates that Intel MKL should determine the amount of the work for the specified device.

Description

If you are using Intel MKL in Automatic Offload mode, the `mkl_mic_set_workdivision` function specifies how much work each Intel Xeon Phi coprocessor or the host CPU should do. This function takes precedence over the `MKL_HOST_WORKDIVISION`, `MKL_MIC_WORKDIVISION`, and `MKL_MIC_<number>_WORKDIVISION` environment variables (see the *Intel MKL User's Guide* for details).

- Intel MKL interprets the fraction of work set by the `mkl_mic_set_workdivision` function as guidance toward dividing work between coprocessors, but the library may choose a different work division if necessary.
- Intel MKL resolves the collision that arises if the sum of all the work-division fractions does not equal one.
- For LAPACK routines, setting the fraction of work to any value other than 0.0 enables the specified processor for Automatic Offload mode. However Intel MKL LAPACK does not use the value specified to divide the workload. For example, setting the fraction to 0.5 has the same effect as setting the fraction to 1.0.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	Result status: = 0 Indicates that the fraction is successfully set. < 0 Indicates a failure to set the fraction.

See Also

[mkl_mic_get_device_count](#)

[mkl_mic_enable](#)

[mkl_mic_get_workdivision](#)

[mkl_mic_get_cpuinfo](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_workdivision

For computations in the Automatic Offload mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do.

Syntax

```
ierr = mkl_mic_get_workdivision( target_type, target_number, wd )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use one of the following values:

Name	Type	Description
<i>target_number</i>	INTEGER*4	<ul style="list-style-type: none"> • MKL_TARGET_HOST - host CPU • MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default <p>The device to retrieve the fraction of work for. Takes the following values:</p> <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <i>target_number</i> = 6 determines the coprocessor number 2. • < 0. Reserved. <p>If <i>target_type</i> = MKL_TARGET_HOST, the function ignores the <i>target_number</i> parameter, which may have any value.</p>

Output Parameters

Name	Type	Description
<i>wd</i>	REAL*8	<p>The fractional amount of the work that the specified device should do, where $0.0 \leq wd \leq 1.0$.</p> <p>MKL_MIC_AUTO_WORKDIVISION set for <i>wd</i> indicates that Intel MKL should determine the amount of the work for the specified device.</p>

Description

If you are using Intel MKL in the Automatic Offload mode, the `mkl_mic_get_workdivision` function provides you with the amount of the work that the specified coprocessor or host CPU is configured to do.

Return Values

Name	Type	Description
<i>ierr</i>	INTEGER*4	<p>Result status:</p> <ul style="list-style-type: none"> = 0 Indicates that the fraction is successfully returned. < 0 Indicates a failure to retrieve the fraction.

See Also

[mkl_mic_get_device_count](#)
[mkl_mic_enable](#)
[mkl_mic_set_workdivision](#)
[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_set_max_memory

Sets the maximum amount of Intel Xeon Phi coprocessor memory reserved for the Automatic Offload computations.

Syntax

```
ierr = mkl_mic_set_max_memory( target_type, target_number, mem_size )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use the value of <code>MKL_TARGET_MIC</code> - Intel Xeon Phi coprocessor, default.
<code>target_number</code>	INTEGER*4	The coprocessor number for which the maximum memory used for Automatic Offload computations is set. Takes the following values: <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2. • < 0. Reserved.
<code>mem_size</code>	INTEGER*8	For the <code>target_number</code> device, the amount of memory in kilobytes to reserve for Automatic Offload computations. Intel MKL attempts to not exceed the specified memory size for Automatic Offload computations.

Description

The `mkl_mic_set_max_memory` function enables you to limit coprocessor memory used by the Automatic Offload computations. Intel MKL reserves the specified memory on Intel Xeon Phi coprocessors. This can improve the performance of Automatic Offload computations by reducing the cost of buffer initialization and data transfer. The specified memory is reserved for the calling process, and the threads of a process share the specified memory. Intel MKL allocates additional memory for each process that performs Automatic Offload computations.

This function takes precedence over the `MKL_MIC_MAX_MEMORY` and `MKL_MIC_<number>_MAX_MEMORY` environment variables (see the *Intel MKL User's Guide* for details).

- Call `mkl_mic_set_max_memory` before any Intel MKL functions that do Automatic Offload computations.
 - Use `mkl_mic_free_memory` to free the coprocessor memory reserved for the Automatic Offload computations.
-

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	<p>Result status:</p> <ul style="list-style-type: none"> = 0 Indicates the coprocessor memory reserved for Automatic Offload is set successfully. < 0 Indicates an error.

See Also

[mkl_mic_free_memory](#)

[mkl_mic_enable](#)

[mkl_mic_get_meminfo](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_free_memory

Frees the coprocessor memory reserved for the Automatic Offload computations.

Syntax

```
ierr = mkl_mic_free_memory( target_type, target_number )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use the value of <code>MKL_TARGET_MIC</code> - Intel Xeon Phi coprocessor, default.
<code>target_number</code>	INTEGER*4	<p>The coprocessor number for which the memory reserved for Automatic Offload computations is freed. Takes the following values:</p> <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2.

Name	Type	Description
		<ul style="list-style-type: none"> <0. Reserved.

Description

The `mkl_mic_free_memory` function frees the coprocessor memory reserved for the Automatic Offload computations. If you call `mkl_mic_set_max_memory` to specify the maximum coprocessor memory for Automatic Offload computations, Intel MKL reserves and reuses the specified coprocessor memory during multiple Automatic Offload calls. You can reclaim the coprocessor memory by calling `mkl_mic_free_memory`.

- Currently, Intel MKL reserves the coprocessor memory only if the `mkl_mic_set_max_memory` function is called. Therefore, `mkl_mic_free_memory` has no effect unless there is a prior call to the `mkl_mic_set_max_memory` function.
- If you do not call `mkl_mic_free_memory`, Intel MKL frees the coprocessor memory at the program exit.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	Result status: = 0 Indicates the coprocessor memory reserved for Automatic Offload is freed successfully. < 0 Indicates an error.

See Also

[mkl_mic_set_max_memory](#)

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_register_memory

Enables/disables the `mkl_malloc` function running in Automatic Offload mode to register allocated memory.

Syntax

```
call mkl_mic_register_memory( control )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>control</code>	INTEGER*4	Desired behavior of <code>mkl_malloc</code> . Possible values: 0 - not register allocated memory, default.

Name	Type	Description
		1 - register allocated memory if Automatic Offload (AO) mode is enabled.

Description

The `mkl_mic_register_memory` function enables or disables the `mkl_malloc` function to register allocated memory when `mkl_malloc` runs in AO mode.

Registration of memory may reduce the overhead introduced by the operating system during data transfers between Intel Xeon Phi coprocessors and the host CPU.

This function takes precedence over the `MKL_MIC_REGISTER_MEMORY` environment variable.

If AO mode is disabled, the function has no effect.

See Also

- `mkl_malloc` Allocates an aligned memory buffer.
- `mkl_mic_enable` Enables Automatic Offload mode.
- [Using a Fortran Interface Module for Support Functions](#)

`mkl_mic_set_device_num_threads`

Sets the maximum number of OpenMP threads to use on an Intel Xeon Phi coprocessor for the Automatic Offload computations.*

Syntax

```
ierr = mkl_mic_set_device_num_threads( target_type, target_number, num_threads )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use the value of <code>MKL_TARGET_MIC</code> - Intel Xeon Phi coprocessor, default.
<code>target_number</code>	INTEGER*4	The coprocessor number for which the maximum number of threads to be used for Automatic Offload computations is set. Takes the following values: <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2.

Name	Type	Description
<code>num_threads</code>	INTEGER*4	<ul style="list-style-type: none"> • <0. Reserved. <p>The number of threads to use for Automatic Offload computations on the <code>target_number</code> coprocessor. Must be greater than 0.</p>

Description

The `mkl_mic_set_device_num_threads` function enables you to limit the number of OpenMP threads to use for Automatic Offload computations on a specific Intel Xeon Phi coprocessor.

This function takes precedence over the `MIC_OMP_NUM_THREADS` and `MKL_MIC_<number>_OMP_NUM_THREADS` environment variables (see the *Intel MKL User's Guide* for details).

Unless the maximum number of threads is set by the environment variables or this function, Intel MKL uses all coprocessor cores.

Call `mkl_mic_set_device_num_threads` before initialization of Automatic Offload mode. Otherwise the function returns -1 without setting the number of threads.

NOTE

Automatic Offload mode can be initialized as follows:

- *explicitly*, in a call to the `mkl_mic_enable` function.
 - *implicitly*, in the first call to an Intel MKL function that does Automatic Offload computations (for example: `?GEMM`). See the *Intel MKL Release Notes* for which functions support Automatic Offload mode.
-

The `mkl_mic_set_device_num_threads` function sets the maximum number of threads to use for the Automatic Offload computations on Intel Xeon Phi coprocessors only. To control host CPU threading, use general threading control functions and see "Using Additional Threading Control" in the *Intel MKL User's Guide* for more information.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	<p>Result status:</p> <ul style="list-style-type: none"> = 0 Indicates that the number of threads is set successfully. < 0 Indicates a failure to set the number of threads.

See Also

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions Threading Control](#)

mkl_mic_set_resource_limit

For computations in the Automatic Offload mode, sets the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.

Syntax

```
ierr = mkl_mic_set_resource_limit( fraction )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>fraction</i>	REAL*8	<p>The fractional amount of Intel Xeon Phi coprocessor computational resources that the calling process can use for Automatic Offload (AO).</p> <p>Possible values: $0.0 \leq \textit{fraction} \leq 1.0$.</p> <p>Special values:</p> <ul style="list-style-type: none"> • 0.0 - you need to manage the coprocessors explicitly using the mkl_mic_set_device_num_threads function or the <code>MIC_KMP_AFFINITY</code> environment variable. Default. • <code>MKL_MPI_PPN</code> constant - enables a fully automated mode for message-passing interface (MPI) applications. In this mode, Intel MKL tries to read the number of MPI processes per node (<i>ppn</i>) from the environment variables passed to the process by MPI. If this attempt is successful, Intel MKL sets the actual value of the fractional amount to $1.0/\textit{ppn}$.

NOTE

For Intel® MPI Library, Open MPI, and IBM Platform MPI, Intel MKL automatically detects *ppn*. For other MPI implementations, use the `MKL_MPI_PPN` environment variable to set *ppn*.

Description

If you are using Intel MKL in the AO mode, the `mkl_mic_set_resource_limit` function specifies how much of the computational resources of Intel Xeon Phi coprocessors can be used by the calling process. Use this function if you need to share coprocessor cores automatically across multiple processes that call Intel MKL in the AO mode. For example, this might be useful in MPI applications.

You can also enable this functionality using the `MKL_MIC_RESOURCE_LIMIT` environment variable (see the *Intel MKL User's Guide* for details), but the `mkl_mic_set_resource_limit` function take precedence over the environment variable.

If *fraction* is set to a valid non-zero value, Intel MKL enables automatic reservation of Intel Xeon Phi coprocessor cores.

Actual reservation is made during a call to an Intel MKL AO function and works as follows:

1. Intel MKL converts *fraction* to a number of Intel Xeon Phi coprocessor cores and tries to find cores that are not reserved by other processes.

The number of available cores can be less than the requested number. Intel MKL considers cores of all available coprocessors as a single space and can find cores on different coprocessors. However, Intel MKL tends to reserve as many cores as were requested and to reserve cores compactly (using as few coprocessors as possible).

- Intel MKL exclusively reserves cores found in the previous step.

Intel MKL makes reservation safely across a persistent shared memory region using semaphores.

- Intel MKL performs AO computations on reserved cores only.

You do not need to set any threading parameters for Intel Xeon Phi coprocessors, such as `MIC_OMP_NUM_THREADS` or `MIC_KMP_AFFINITY` (these settings are ignored when automatic resource sharing is enabled).

- Upon completion of an AO call, Intel MKL releases reserved cores.

Be aware of the following features of the automatic reservation:

- Execution of Intel MKL AO functions falls back to the host if Intel MKL fails to reserve the minimum number of Intel Xeon Phi coprocessor cores that provide a benefit over executing on the host.
- The last core of Intel Xeon Phi coprocessors cannot be reserved because it is used for system processes.
- Reservation works only within a space of a single user. In other words, Intel MKL cannot share Intel Xeon Phi coprocessor resources across processes from different users.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

- One Intel Xeon Phi coprocessor with 61 cores is available on the system. One process calls `mkl_mic_set_resource_limit(1.0)` and then calls `dgemm` in AO mode. As a result, 60 Intel Xeon Phi coprocessor cores are reserved at the beginning of AO `dgemm`. Then `mkl_mic_set_resource_limit(0.3)` is invoked and AO `dgemm` is called one more time. As a result, `dgemm` is run on 18 Intel Xeon Phi coprocessor cores. During this run of AO `dgemm`, another process calls `mkl_mic_set_resource_limit(1.0)` and invokes AO `dgetrf`. Because 18 cores are reserved by the first process, only 42 cores are reserved for AO `dgetrf`.
- Two Intel Xeon Phi coprocessors with 61 cores each are available on the system. Three processes simultaneously call `mkl_mic_set_resource_limit(0.34)` and then call `dpotrf` in AO mode. As a result, one process receives 40 cores from coprocessor 1, another process receives 40 cores from coprocessor 2, and the remaining process is given 20 cores from each of the two coprocessors for a total of 40 cores.
- Two Intel Xeon Phi coprocessors with 58 cores each are available on the system. The user sets `MKL_MIC_PPN=4` and runs an MPI application with 4 MPI ranks. The following sequence of functions is called in each MPI process:

```
mkl_mic_set_workdivision(MKL_TARGET_MIC, 2, 0.0);
mkl_mic_set_resource_limit(MKL_MIC_PPN);
MPI_Init(...);
//calls to Intel MKL AO functions;
MPI_Finalize(...);
```

As a result, actual *fraction* is set to 0.25 for each MPI process, and each MPI process receives 14 cores on Intel Xeon Phi coprocessor 1 (because the user excluded co-processor 2 from computations by setting zero workdivision for it).

Return Values

Name	Type	Description
<i>ierr</i>	INTEGER*4	Result status: = 0 Indicates that the fraction is set successfully. < 0 Indicates a failure to set the fraction.

See Also

- [mkl_mic_get_resource_limit](#)
- [mkl_mic_enable](#)
- [mkl_mic_get_device_count](#)
- [Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_resource_limit

For computations in the Automatic Offload mode, retrieves the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.

Syntax

```
ierr = mkl_mic_get_resource_limit( fraction )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Output Parameters

Name	Type	Description
<i>fraction</i>	REAL*8	The fractional amount of Intel Xeon Phi coprocessor computational resources that the calling process can use for Automatic Offload (AO). Possible values: $0.0 \leq \textit{fraction} \leq 1.0$.

Description

If you are using Intel MKL in the AO mode, the `mkl_mic_get_resource_limit` function retrieves the fractional amount of Intel Xeon Phi coprocessor computational resources that can be used by the calling process.

Return Values

Name	Type	Description
<i>ierr</i>	INTEGER*4	Result status: = 0 Indicates that the fraction is successfully returned.

Name	Type	Description
		< 0 Indicates a failure to return the fraction.

See Also

[mkl_mic_set_resource_limit](#)

[mkl_mic_enable](#)

[mkl_mic_get_device_count](#)

[mkl_mic_get_workdivision](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_set_offload_report

Turns on/off reporting of Automatic Offload profiling.

Syntax

```
iprev = mkl_mic_set_offload_report( enabled )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>enabled</i>	INTEGER*4	This parameter specifies whether to turn the reporting on or off. Takes values with boolean semantics: <ul style="list-style-type: none"> • 0 - Reporting of Automatic Offload profiling should be turned off. • 1 - Reporting of Automatic Offload profiling should be turned on.

Description

If the `OFFLOAD_REPORT` environment variable has the value of 1 or 2, the `mkl_mic_set_offload_report` function turns on/off reporting of the Automatic Offload profiling at run time. If the `OFFLOAD_REPORT` environment variable is not set or is set to a value different from 1 and 2, the function has no effect. If `OFFLOAD_REPORT` is set to 1 or 2, the reporting is turned on at a program startup. The `mkl_mic_set_offload_report` function does not change the reporting level or the value of the environment variable. Instead, it determines which Intel MKL functions called in the Automatic Offload mode produce profiling reports.

For details of the reporting level and `OFFLOAD_REPORT` environment variable, see the *Intel MKL User's Guide*.

Return Values

Name	Type	Description
<i>iprev</i>	INTEGER*4	The previous value that determined whether the reporting was on or off.

mkl_mic_set_flags

Sets flags to control the behavior of computations in the Automatic Offload mode.

Syntax

```
old_flags = mkl_mic_set_flags( flags )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>flags</code>	INTEGER*4	<p>The settings requested:</p> <ul style="list-style-type: none"> • <code>MKL_MIC_DEFAULT_FLAGS</code> - The default Automatic Offload (AO) mode, when computations fall back to host in the case of any AO error, such as lack of memory or other resources on the Intel Xeon Phi coprocessor. • <code>MKL_MIC_DISABLE_HOST_FALLBACK</code> - The mode when computations do not fall back to host in the case of an AO error, but the AO function immediately returns control to the calling program without completing the computations.

Description

This function sets flags to control the behavior of computations in the AO mode. Depending on the flags set, the AO program may silently return to the calling program in the case of an AO error. To check whether the AO call is successful, call the `mkl_mic_get_status` function and check whether the return value is non-negative. For more details of `mkl_mic_get_status` and in particular, for a sequence of function calls that enables checking the status of an AO call, see [mkl_mic_get_status](#).

Return Values

Name	Type	Description
<code>old_flags</code>	INTEGER*4	The previously set flags.

See Also

[mkl_mic_get_flags](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_flags

Retrieves flags that control the behavior of computations in the Automatic Offload mode.

Syntax

```
flags = mkl_mic_get_flags()
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function retrieves flags that control the behavior of computations in the Automatic Offload (AO) mode. These flags are either the default flags for the AO mode or the flags set in a call to the `mkl_mic_set_flags` function.

NOTE

Depending on the flags set, the AO program may silently return to the calling program in the case of an AO error. To check whether the AO call is successful, call the `mkl_mic_get_status` function and check whether the return value is non-negative. For more details of `mkl_mic_get_status` and in particular, for a sequence of function calls that enables checking the status of an AO call, see [mkl_mic_get_status](#).

Return Values

Name	Type	Description
<code>flags</code>	INTEGER*4	<p>The flags returned, which can be one of the following:</p> <ul style="list-style-type: none"> • <code>MKL_MIC_DEFAULT_FLAGS</code> - The default AO mode, when computations fall back to host in the case of any AO error, such as lack of memory or other resources on the Intel Xeon Phi coprocessor. • <code>MKL_MIC_DISABLE_HOST_FALLBACK</code> - The mode when computations do not fall back to host in the case of an AO error, but the AO function immediately returns control to the calling program without completing the computations.

See Also

[mkl_mic_set_flags](#)

[Using a Fortran Interface Module for Support Functions](#)

`mkl_mic_get_status`

For the Automatic Offload mode, returns the status of the latest call to an Intel MKL function.

Syntax

```
status = mkl_mic_get_status( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`

- Module (source): `mkl_service.f90`

Description

This function returns the status of the latest call to an Intel MKL function done in the Automatic Offload (AO) mode. The sign of the returned value characterizes the status of the AO computations at a high level:

- = 0 - The computations completed successfully using AO
- > 0 - The computations were done without using AO
- < 0 - The computations were not completed at all

In an AO call to an Intel MKL function, offloading computations as expected may not be possible for various reasons, such as:

- Lack of available memory or computation cores on an Intel Xeon Phi coprocessor
- Incorrect environment settings (which result in a failure to locate or load a coprocessor driver or libraries)
- Communication error during data transfer to or from a coprocessor (for example, a remote process dies unexpectedly)
- Other unexpected error

The status returned helps you to find out the reason of offload failure.

Because the status is a thread-local value, to be able to check the status of an Intel MKL AO function, the following sequence of function calls in the same thread is required:

1. `mkl_mic_clear_status`
2. Intel MKL function
3. `mkl_mic_get_status`

Return Values

Name	Type	Description
<code>status</code>	INTEGER*4	<p>The thread-local value of the status:</p> <ul style="list-style-type: none"> • <code>MKL_MIC_SUCCESS</code> (0) - The computations successfully completed with offloading to Intel Xeon Phi coprocessors. • <code>MKL_MIC_NOT_IMPL</code> (1) - The computations successfully completed only on the host because AO is not implemented for this function or for the combination of input parameters of the function. • <code>MKL_MIC_HOST_FALLBACK</code> (2) - The computations successfully completed only on the host because AO could not start. Possible reasons: the environment path does not include required system libraries, resources are insufficient, memory allocation failed on a coprocessor, and so on. • <code>MKL_MIC_DISABLED</code> (3) - The computations successfully completed only on the host because AO is disabled. • <code>MKL_MIC_FAILED</code> (-1) - The computations were not completed. AO could start, but an error occurred during the computations. Note that the function output data may be corrupted in this case. • <code>MKL_MIC_HOST_FALLBACK_DISABLED</code> (-2) - The computations were not completed because AO could not start. Possible reasons: the environment path does not include required system libraries, resources are

Name	Type	Description
		insufficient, memory allocation failed on a coprocessor, and so on. Unlike <code>MKL_MIC_HOST_FALLBACK</code> , the <code>mkl_mic_get_status</code> function can return this status only if the <code>mkl_mic_set_flags</code> function disabled falling back to host. Unlike for <code>MKL_MIC_FAILED</code> status, the function input and output data remain untouched.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[mkl_mic_set_flags](#)

[mkl_mic_clear_status](#)

[Using a Fortran Interface Module for Support Functions](#)

`mkl_mic_clear_status`

For the Automatic Offload mode, clears the status of the latest call to an Intel MKL function.

Syntax

```
status = mkl_mic_clear_status( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function clears the status of the latest call to an Intel MKL function done in the Automatic Offload (AO) mode. Call `mkl_mic_clear_status` before calling an Intel MKL function in the AO mode to be able to check the status of the AO call. Because the status is a thread-local value, getting the status for an Intel MKL function requires the following sequence of function calls in the same thread:

1. `mkl_mic_clear_status`
2. Intel MKL function
3. `mkl_mic_get_status`

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[mkl_mic_get_status](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_meminfo

Retrieves the amount of total and free memory for the specified coprocessor or host CPU.

Syntax

```
ierr = mkl_mic_get_meminfo( target_type, target_number, totalmem, freemem )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use one of the following values: <ul style="list-style-type: none"> • MKL_TARGET_HOST - host CPU • MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default
<code>target_number</code>	INTEGER*4	The device to retrieve the memory information for. Takes the following values: <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2. • < 0. Reserved.

Name	Type	Description
		If <code>target_type = MKL_TARGET_HOST</code> , the function ignores the <code>target_number</code> parameter, which may have any value.

Output Parameters

Name	Type	Description
<code>totalmem</code>	INTEGER*4	The total amount of memory on the target device in kilobytes.
<code>freemem</code>	INTEGER*4	The amount of free memory available on the target device in kilobytes.

Description

Use this function to retrieve the amount of total and free memory (in kilobytes) available on an offload device or host CPU.

Return Values

Name	Type	Description
<code>ierr</code>	INTEGER*4	Result status: = 0 Indicates that the memory information for the target device is successfully returned. < 0 Indicates a failure to return the information.

See Also

[mkl_mic_set_max_memory](#)

[mkl_mic_get_cpuserinfo](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_cpuserinfo

Retrieves the number of cores, hardware threads, and frequency for the specified coprocessor or host CPU.

Syntax

```
ierr = mkl_mic_get_cpuserinfo( target_type, target_number, ncores, nthreads, freq )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>target_type</code>	INTEGER*4	Type of the target device. Use one of the following values:

Name	Type	Description
<i>target_number</i>	INTEGER*4	<ul style="list-style-type: none"> MKL_TARGET_HOST - host CPU MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default <p>The device to retrieve the information for. Takes the following values:</p> <ul style="list-style-type: none"> ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <i>target_number</i> = 6 determines the coprocessor number 2. < 0. Reserved. <p>If <i>target_type</i> = MKL_TARGET_HOST, the function ignores the <i>target_number</i> parameter, which may have any value.</p>

Output Parameters

Name	Type	Description
<i>ncores</i>	INTEGER*4	The number of physical cores on the target device.
<i>nthreads</i>	INTEGER*4	The number of hardware threads on the target device.
<i>freq</i>	REAL*8	The frequency in Hz of the target device.

Description

Use this function to retrieve the number of cores, hardware threads, and frequency for the host CPU or an Intel Xeon Phi coprocessor.

Return Values

Name	Type	Description
<i>ierr</i>	INTEGER*4	<p>Result status:</p> <ul style="list-style-type: none"> = 0 Indicates that the information for the target device is successfully returned. < 0 Indicates a failure to return the information.

See Also

[mkl_mic_get_meminfo](#)
[mkl_mic_get_device_count](#)
[mkl_mic_set_resource_limit](#)
[mkl_mic_set_workdivision](#)
[Using a Fortran Interface Module for Support Functions](#)

Conditional Numerical Reproducibility Control

The CNR mode of Intel MKL ensures bitwise reproducible results from run to run of Intel MKL functions on a fixed number of threads for a specific Intel instruction set architecture (ISA) under the following conditions:

- Calls to Intel MKL occur in a single executable
- The number of computational threads used by the library does not change in the run

Intel MKL offers both functions and environment variables to support conditional numerical reproducibility. See the *Intel MKL User's Guide* for more information on bitwise reproducible results of computations and for details about the environment variables.

The support functions enable you to configure the CNR mode and also provide information on the current and optimal CNR branch on your system. [Usage Examples for CNR Support Functions](#) illustrate usage of these functions.

Important

Call the functions that define the behavior of CNR before any of the math library functions that they control.

Intel MKL provides named constants for use as input and output parameters of the functions instead of integer values. See [Named Constants for CNR Control](#) for a list of the named constants.

Although you can configure the CNR mode using either the support functions or the environment variables, the functions offer more flexible configuration and control than the environment variables. Settings specified by the functions take precedence over the settings specified by the environment variables.

Use Intel MKL in the CNR mode only in case a need for bitwise reproducible results is critical. Otherwise, run Intel MKL as usual to avoid performance degradation.

While you can supply unaligned input and output data to Intel MKL functions running in the CNR mode, use of aligned data is recommended. Refer to [Reproducibility Conditions](#) for more details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`mkl_cbwr_set`

Configures the CNR mode of Intel MKL.

Syntax

```
status = mkl_cbwr_set( setting )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>setting</i>	INTEGER*4	CNR branch to set. See Named Constants for CNR Control for a list of named constants that specify the settings.

Description

The `mkl_cbwr_set` function configures the CNR mode. In this release, it sets the CNR branch and turns on the CNR mode.

NOTE

Settings specified by the `mkl_cbwr_set` function take precedence over the settings specified by the `MKL_CBWR` environment variable.

Return Values

Name	Type	Description
<i>status</i>	INTEGER*4	<p>The status of the function completion:</p> <ul style="list-style-type: none"> • <code>MKL_CBWR_SUCCESS</code> - the function completed successfully. • <code>MKL_CBWR_ERR_INVALID_INPUT</code> - an invalid setting is requested. • <code>MKL_CBWR_ERR_UNSUPPORTED_BRANCH</code> - the input value of the branch does not match the instruction set architecture (ISA) of your system. See Named Constants for CNR Control for more details. • <code>MKL_CBWR_ERR_MODE_CHANGE_FAILURE</code> - the <code>mkl_cbwr_set</code> function requested to change the current CNR branch after a call to some Intel MKL function other than a CNR function.

See Also

[Usage Examples for CNR Support Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

`mkl_cbwr_get`

Returns the current CNR settings.

Syntax

```
setting = mkl_cbwr_get( option )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>option</i>	INTEGER*4	<p>Specifies what kind of settings is requested. Named constants define possible values of <i>option</i>:</p> <ul style="list-style-type: none"> • MKL_CBWR_BRANCH - indicates the setting of the CNR branch. • MKL_CBWR_ALL - indicates all CNR-related settings. <hr/> <p>NOTE This release supports no CNR settings other than branch.</p>

Description

The `mkl_cbwr_get` function returns the requested CNR settings. If the function completes successfully and the CNR mode is turned on, `mkl_cbwr_get` returns the *specific* CNR branch.

NOTE

To turn the CNR mode on, use the `mkl_cbwr_set` function or environment variables. For more details, see the *Intel MKL User's Guide*.

Return Values

Name	Type	Description
<i>setting</i>	INTEGER*4	<p>Requested CNR settings. See Named Constants for CNR Control for a list of named constants that specify the settings.</p> <p>If the value of the <i>option</i> parameter is not permitted, contains the MKL_CBWR_ERR_INVALID_INPUT error code.</p>

See Also

Usage Examples for CNR Support Functions
[mkl_cbwr_set](#)

`mkl_cbwr_get_auto_branch`

Automatically detects the CNR code branch for your platform.

Syntax

```
setting = mkl_cbwr_get_auto_branch( )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

The `mkl_cbwr_get_auto_branch` function uses a run-time CPU check to return a CNR branch that is optimized for the processor where the program is currently running.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

Name	Type	Description
<code>setting</code>	INTEGER*4	Automatically detected CNR branch. May be any <i>specific</i> branch listed in Named Constants for CNR Control .

See Also

[Usage Examples for CNR Support Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

Named Constants for CNR Control

Intel MKL provides reproducible results for a certain code branch, determined by the instruction set architecture (ISA). To define CNR code branches, use the following named constants as input/output for conditional numerical reproducibility support functions. Pass named constants to the functions instead of their values.

Named Constant	Value	Description
<code>MKL_CBWR_AUTO</code>	2	CNR mode uses the standard ISA-based dispatching model while ensuring fixed cache sizes, deterministic reductions, and static scheduling CNR mode uses the branch for the following ISA:
<code>MKL_CBWR_COMPATIBLE</code>	3	Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without <code>rcpps/rsqrtps</code> instructions
<code>MKL_CBWR_SSE2</code>	4	Intel SSE2
<code>MKL_CBWR_SSE3</code>	5	DEPRECATED. Intel® Streaming SIMD Extensions 3 (Intel® SSE3). This setting is kept for backward compatibility and is equivalent to <code>MKL_CBWR_SSE2</code> .
<code>MKL_CBWR_SSSE3</code>	6	Supplemental Streaming SIMD Extensions 3 (SSSE3)
<code>MKL_CBWR_SSE4_1</code>	7	Intel® Streaming SIMD Extensions 4-1 (SSE4-1)
<code>MKL_CBWR_SSE4_2</code>	8	Intel® Streaming SIMD Extensions 4-2 (SSE4-2)

Named Constant	Value	Description
MKL_CBWR_AVX	9	Intel® Advanced Vector Extensions (Intel® AVX)
MKL_CBWR_AVX2	10	Intel® Advanced Vector Extensions 2 (Intel® AVX2)

When specifying the CNR branch with the named constants, be aware of the following:

- Reproducible results are provided under [Reproducibility Conditions](#).
- Settings other than `MKL_CBWR_AUTO` or `MKL_CBWR_COMPATIBLE` are available only for Intel processors.
- Intel and Intel compatible CPUs have a few instructions, such as approximation instructions `rcpps/rsqrtps`, that may return different results. Setting the branch to `MKL_CBWR_COMPATIBLE` ensures that Intel MKL does not use these instructions and forces a single Intel SSE2 only code path to be executed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[Usage Examples for CNR Support Functions](#)

Reproducibility Conditions

To get reproducible results from run to run, ensure that the number of threads is fixed and constant. Specifically:

- If you are running your program with OpenMP* parallelization on different processors, explicitly specify the number of threads.
- To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set `MKL_DYNAMIC` and `OMP_DYNAMIC` to `FALSE`. This is especially needed if you are running your program on different systems.
- If you are running your program with the Intel® Threading Building Blocks parallelization, numerical reproducibility is not guaranteed.
 - As usual, you should align your data, even in CNR mode, to obtain the best possible performance. While CNR mode also fully supports unaligned input and output data, the use of it might reduce the performance of some Intel MKL functions on earlier Intel processors. To ensure proper alignment of arrays, allocate memory for them using `mkl_malloc/mkl_calloc`.
 - Conditional Numerical Reproducibility does not ensure that bitwise-identical NaN values are generated when the input data contains NaN values.
 - If dynamic memory allocation fails on one run but succeeds on another run, you may fail to get reproducible results between these two runs.

See Also

[mkl_malloc](#)

[mkl_calloc](#)

Usage Examples for CNR Support Functions

The following examples illustrate usage of support functions for conditional numerical reproducibility.

Setting Automatically Detected CNR Branch

```
PROGRAM MAIN
  INCLUDE 'mkl.fi'
  INTEGER*4 MY_CBWR_BRANCH
C Find the available MKL_CBWR_BRANCH automatically
  MY_CBWR_BRANCH = MKL_CBWR_GET_AUTO_BRANCH()
C User code without Intel MKL calls
C Piece of the code where CNR of Intel MKL is needed
C The performance of Intel MKL functions might be reduced for CNR mode
  IF (MKL_CBWR_SET (MY_CBWR_BRANCH) .NE. MKL_CBWR_SUCCESS) THEN
    PRINT *, 'Error in setting MKL_CBWR_BRANCH! Aborting...'
    RETURN
  ENDIF
C CNR calls to Intel MKL + any other code
END
```

Use of the mkl_cbwr_get Function

```
PROGRAM MAIN
  INCLUDE 'mkl.fi'
  INTEGER*4 MY_CBWR_BRANCH
C Piece of the code where CNR of Intel MKL is analyzed
  MY_CBWR_BRANCH = MKL_CBWR_GET(MKL_CBWR_BRANCH)
  IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_AUTO) THEN
C actions in case of automatic mode
  ELSE IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_SSSE3) THEN
C actions for SSSE3 code
  ELSE
C all other cases
  ENDIF
C User code
END
```

Miscellaneous

mkl_progress

Provides progress information.

Syntax

```
stopflag = mkl_progress( thread, step, stage )
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

Input Parameters

Name	Type	Description
<i>thread</i>	INTEGER*4	The number of the OpenMP* thread the progress routine is called from. 0 is passed for sequential code.
<i>step</i>	INTEGER*4	The linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.
<i>stage</i>	CHARACTER* (*)	Message indicating the name of the routine or the name of the computation stage the progress routine is called from.

Description

The `mkl_progress` function is intended to track progress of a lengthy computation and/or interrupt the computation. By default this routine does nothing but the user application can redefine it to obtain the computation progress information. You can set it to perform certain operations during the routine computation, for instance, to print a progress indicator. A non-zero return value may be supplied by the redefined function to break the computation.

The progress function `mkl_progress` is regularly called from some LAPACK and DSS/PARDISO functions during the computation. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not. If a LAPACK function returns `info=-1002`, the function was interrupted by `mkl_progress`.

WARNING

The `mkl_progress` function only supports OpenMP* threading and sequential execution.

Return Values

Name	Type	Description
<i>stopflag</i>	INTEGER	The stopping flag. A non-zero flag forces the routine to be interrupted. The zero flag is the default return value.

Example

The following example prints the progress information to the standard output device:

```
integer function mkl_progress( thread, step, stage )
integer*4 thread, step
character*(*) stage
print*, 'Thread:', thread, ', stage:', stage, ', step:', step
mkl_progress = 0
return
end
```

mkl_enable_instructions

Enables dispatching for new Intel® architectures.

Syntax

```
irc = mkl_enable_instructions(isa)
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>isa</code>	INTEGER*4	<p>The Intel instruction-set architecture (ISA) for which to enable dispatching.</p> <p>Possible values:</p> <p><code>MKL_ENABLE_AVX512</code> - Requests to enable dispatching Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon® processors.</p>

For more information about the Intel AVX-512 instructions, refer to <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>.

Description

If automatic dispatching is not yet enabled for the specified ISA, `mkl_enable_instructions` enables Intel MKL to dispatch the instruction set to Intel® processors when the library runs on hardware (or simulation) that supports this instruction set.

You can also enable dispatching of new Intel architectures using the `MKL_ENABLE_INSTRUCTIONS` environment variable (for more details, see the *Intel MKL User's Guide*), but the settings specified by the `mkl_enable_instructions` function take precedence over the settings specified by the environment variable.

Return Values

Name	Type	Description
<code>irc</code>	INTEGER*4	<p>Usage status of the specified instruction set:</p> <p>1 - Intel MKL uses the specified instruction set if the hardware supports it.</p> <p>0 - The request is rejected. Most likely, <code>mkl_enable_instructions</code> has been called after another Intel MKL function.</p>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_set_env_mode

Sets up the mode that ignores environment settings specific to Intel MKL.

Syntax

```
current_mode = mkl_set_env_mode( mode )
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>mode</code>	INTEGER*4	Specifies what mode to set. For details, see Description . Possible values: <ul style="list-style-type: none"> • 0 - Do nothing. • Use this value to query the current environment mode. • 1 - Make Intel MKL ignore environment settings specific to the library.

Description

In the default *environment mode*, Intel MKL can control its behavior using environment variables for threading, memory management, Conditional Numerical Reproducibility, automatic offload, and so on. The `mkl_set_env_mode` function sets up the environment mode that ignores all settings specified by Intel MKL environment variables except `MIC_LD_LIBRARY_PATH` and `MKLROOT`.

Return Values

Name	Type	Description
<code>current_mode</code>	INTEGER*4	Environment mode that was used before the function call: <ul style="list-style-type: none"> • 0 - Default • 1 - Ignore environment settings specific to Intel MKL.

mkl_verbose

Enables or disables Intel MKL Verbose mode.

Syntax

```
status = mkl_verbose(enable)
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<code>enable</code>	INTEGER*4	Desired state of the Intel MKL Verbose mode. Indicates whether printing Intel MKL function call information should be turned on or off. Possible values: <ul style="list-style-type: none">• 0 - disable the Verbose mode.• 1 - enable the Verbose mode.

Description

This function enables or disables the Intel MKL Verbose mode, in which computational functions print call description information. For details of the Verbose mode, see the *Intel MKL User's Guide*, available in the Intel® Software Documentation Library.

NOTE

The setting for the Verbose mode specified by the `mkl_verbose` function takes precedence over the setting specified by the `MKL_VERBOSE` environment variable.

Return Values

Name	Type	Description
<code>status</code>	INTEGER*4	<ul style="list-style-type: none">• If the requested operation completed successfully, contains previous state of the verbose mode:<ul style="list-style-type: none">• 0 - disabled• 1 - enabled• If the function failed to complete the operation because of an incorrect input parameter, equals -1.

See Also

[Intel Software Documentation Library](#)

`mkl_set_mpi`

Sets the implementation of the message-passing interface to be used by Intel MKL.

Syntax

```
status = mkl_set_mpi(vendor, custom_library_name)
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Input Parameters

Name	Type	Description
<i>vendor</i>	INTEGER*4	Specifies the implementation of the message-passing interface (MPI) to use: Possible values: <ul style="list-style-type: none"> • MKL_BLACS_CUSTOM - a custom MPI library. Requires a prebuilt custom MPI BLACS library. • MKL_BLACS_MSMPPI - Microsoft MPI library. • MKL_BLACS_INTELMPI - Intel® MPI library. • MKL_BLACS_MPICH - MPICH MPI library.
<i>custom_library_name</i>	CHARACTER*	The filename (without a directory name) of the custom BLACS dynamic library to use. This library must be located in the directory with your application executable or with Intel MKL dynamic libraries. Can be NULL or an empty string.

Description

Call this function to set the MPI implementation to be used by Intel MKL on Windows* OS when dynamic Intel MKL libraries are used. For all other configurations, the function returns an error indicating that you cannot set the MPI implementation. You can specify your own prebuilt dynamic BLACS library for a custom MPI by setting *vendor* to MKL_BLACS_CUSTOM and optionally passing the name of the custom BLACS dynamic library. If the *custom_library_path* parameter is NULL or an empty string, Intel MKL uses the default platform-specific library name: *mkl_blacs_custom_lp64.dll* or *mkl_blacs_custom_ilp64.dll*, depending on whether the BLACS interface linked against your application is LP64 or ILP64.

Return Values

Name	Type	Description
<i>status</i>	INTEGER*4	The return status: <ul style="list-style-type: none"> • 0 - The function completed successfully. • -1 - The <i>vendor</i> parameter is invalid. • -2 - The <i>custom_library_name</i> parameter is invalid. • -3 - The MPI library cannot be set at this point.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_finalize

Terminates Intel MKL execution environment and frees resources allocated by the library.

Syntax

```
call mkl_finalize
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

Description

This function frees resources allocated by Intel MKL. Once this function is called, the application can no longer call Intel MKL functions other than `mkl_finalize`.

In particular, the `mkl_finalize` function enables you to free resources when a third-party shared library is statically linked to Intel MKL. To avoid resource leaks that may happen when a shared library is loaded and unloaded multiple times, call `mkl_finalize` each time the library is unloaded. The recommended method to do this depends on the operating system:

- On Linux* or OS X*, place the call into a shared library destructor.
- On Windows*, call `mkl_finalize` from the `DLL_PROCESS_DETACH` handler of `DllMain`.

NOTE

Intel MKL shared libraries automatically perform finalization when they are unloaded. If an application is statically linked to Intel MKL, the operating system frees all resources allocated by Intel MKL during termination of the process associated with the application.

BLACS Routines

This chapter describes the Intel® Math Kernel Library implementation of FORTRAN 77 routines from the BLACS (Basic Linear Algebra Communication Subprograms) package. These routines are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel MKL intended for the Linux* and Windows* OSs as the communication layer of ScaLAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic [matrix shapes](#) in a special section.

The BLACS routines implemented in Intel MKL are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The [Combines](#) take data distributed over processes and combine the data to produce a result. The [Point to Point](#) routines are intended for point-to-point communication and [Broadcast](#) routines send data possessed by one process to all processes within a scope.

The [Support routines](#) perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of m rows and n columns, with a leading dimension, lda , that determines the distance between successive columns in memory.

The *general rectangular* matrices take the following parameters as input when determining what array to operate on:

- | | |
|-----|---|
| m | (input) INTEGER. The number of matrix rows to be operated on. |
| n | (input) INTEGER. The number of matrix columns to be operated on. |
| a | (input/output) TYPE (depends on routine), array of dimension (lda, n) .
A pointer to the beginning of the (sub)array to be sent. |

lda (input) INTEGER. The distance between two elements in matrix row.

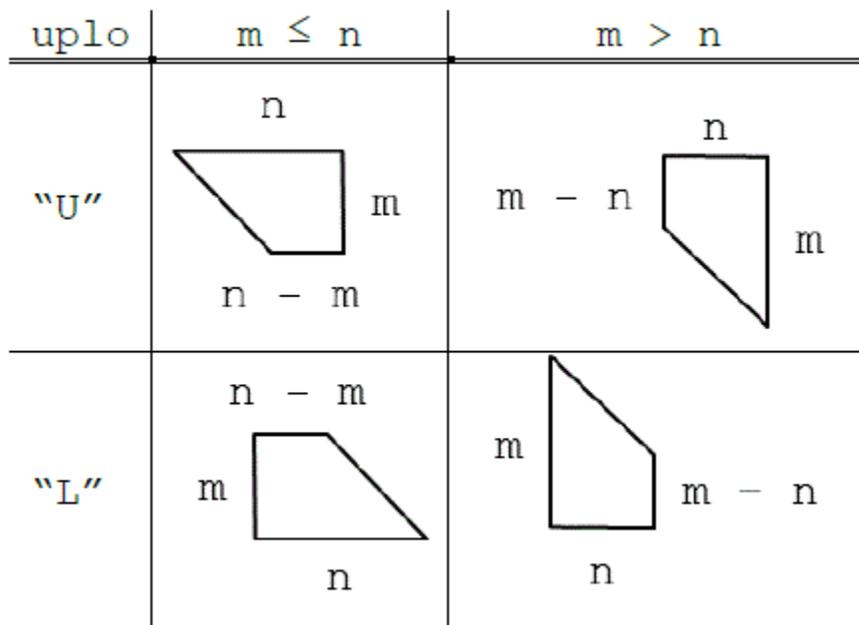
The second class of matrices recognized by the BLACS are *trapezoidal* matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by m , n , and lda , as above, but they have two additional parameters as well. These parameters are:

uplo (input) CHARACTER*1 . Indicates whether the matrix is upper or lower trapezoidal, as discussed below.

diag (input) CHARACTER*1 . Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

Trapezoidal Arrays Shapes



The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

Repeatability and Coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. Since floating point computations are inexact, algorithms are classified according to whether they are *repeatable* and to what degree they guarantee *coherence*.

- Repeatability: a routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input.
- Coherent: a routine is coherent if all processes selected to receive the answer get identical results.

NOTE

Repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. But inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

Repeatability

Because the precision of floating point arithmetic is limited, it is not truly associative: $(a + b) + c$ might not be the same as $a + (b + c)$. The lack of exact arithmetic can cause problems whenever the possibility for reordering of floating point calculations exists. This problem becomes prevalent in parallel computing due to race conditions in message passing. For example, consider a routine which sums numbers stored on different processes. Assume this routine runs on four processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0:0, process 1 has the value 1:0, and so on.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 adds them up, and sends the result back to all processes. So, process 0 would add a number to 0:0 in the first step. If receiving the process numbers is ordered so that process 0 always receives the message from process 1 first, then 2, and finally 3, this results in a repeatable algorithm, which evaluates the expression $((0:0 + 1:0) + 2:0) + 3:0$.

However, to get the best parallel performance, it is better not to require a particular ordering, and just have process 0 add the first available number to its value and continue to do so until all numbers have been added in. Using this method, a race condition occurs, because the order of the operation is determined by the order in which process 0 receives the messages, which can be effected by any number of things. This implementation is not repeatable, because the answer can vary between invocations, even if the input is the same. For instance, one run might produce the sequence $((0:0 + 1:0) + 2:0) + 3:0$, while a subsequent run could produce $((0:0 + 2:0) + 1:0) + 3:0$. Both of these results are correct summations of the given numbers, but because of floating point roundoff, they might be different.

Coherence

A routine produces coherent output if all processes are guaranteed to produce the exact same results. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, no routine is guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms guarantee coherence only if floating point operations are done in the exact same order on every node. This is *homogeneous coherence*: the result will be coherent if the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is *heterogeneous coherence*, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes.

Example of Incoherence

An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. The previous example of summing the process numbers demonstrates this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The calculations performed by each process receives would then be:

- Process 0 : $((0:0 + 1:0) + 2:0) + 3:0$
- Process 1 : $((1:0 + 2:0) + 3:0) + 0:0$
- Process 2 : $((2:0 + 3:0) + 0:0) + 1:0$
- Process 3 : $((3:0 + 0:0) + 1:0) + 0:0$

All of these results are equally valid, and since all the results might be different from each other, this algorithm is incoherent. Notice, however, that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

Example of Homogeneous Coherence

Another way to perform this summation is for all processes to send their data to all other processes, and to ensure the result is not incoherent, enforce the ordering so that the calculation each node performs is $((0:0 + 1:0) + 2:0) + 3:0$. This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since there is a specific ordering to the addition, this algorithm is repeatable.

Example of Heterogeneous Coherence

In the final example, all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can perform the operations in any order and it will give coherent results as long as communication is itself coherent. If a particular order is not forced on the the addition, the algorithm will not be repeatable. If a particular order is forced, it will be repeatable.

Summary

Repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

- **Repeatability:** The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
- **Homogeneous coherence:** All processes selected to possess the result will receive the exact same answer if:
 - Communication does not change the value of the communicated data.
 - All processes perform floating point arithmetic exactly the same.
- **Heterogeneous coherence:** All processes will receive the exact same answer if communication does not change the value of the communicated data.

In general, lack of the associative property for floating point calculations may cause both incoherence and non-repeatability. Algorithms that rely on redundant computations are at best homogeneous coherent, and algorithms in which one process broadcasts the result are heterogeneous coherent. Repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they can usually be ignored. However, in very specific situations, these issues may become very important. A stopping criteria should not be based on incoherent results, for instance. Also, a user creating and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run.

In the BLACS, coherence and repeatability apply only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. Communication is assumed to be coherent, which for communication implies that it is also repeatable.

For combine operations, the BLACS allow you to set flags indicating that you would like combines to be repeatable and/or heterogeneous coherent (see [blacs_get](#) and [blacs_set](#) for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS restrict the topologies which can be used so that one process calculates the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligible to serious depending on the application.

A couple of additional notes are in order. Incoherence and nonrepeatability can arise as a result of floating point errors, as discussed previously. This might lead you to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS treats integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\text{sqr}(r^2 + i^2)$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

BLACS Combine Operations

This section describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes' data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (*max/min/sum*) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4 x 2 array of inputs produces a 4 x 2 answer array.

When the *max/min* comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol ? in the routine names below masks the data type:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex.

BLACS Combines

Routine name	Results of operation
gamx2d	Entries of result matrix will have the value of the greatest absolute value found in that position.
gamn2d	Entries of result matrix will have the value of the smallest absolute value found in that position.

Routine name	Results of operation
<code>gsum2d</code>	Entries of result matrix will have the summation of that position.

?gamx2d

Performs element-wise absolute value maximization.

Syntax

```
call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

<code>icontxt</code>	INTEGER. Integer handle that indicates the context.
<code>scope</code>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<code>top</code>	CHARACTER*1. Communication pattern to use during the combine operation.
<code>m</code>	INTEGER. The number of matrix rows to be combined.
<code>n</code>	INTEGER. The number of matrix columns to be combined.
<code>a</code>	TYPE array (<code>lda</code> , <code>n</code>). Matrix to be compared with to produce the maximum.
<code>lda</code>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<code>rcflag</code>	INTEGER. If <code>rcflag = -1</code> , the arrays <code>ra</code> and <code>ca</code> are not referenced and need not exist. Otherwise, <code>rcflag</code> indicates the leading dimension of these arrays, and so must be $\geq m$.
<code>rdest</code>	INTEGER. The process row coordinate of the process that should receive the result. If <code>rdest</code> or <code>cdest = -1</code> , all processes within the indicated scope receive the answer.
<code>cdest</code>	INTEGER. The process column coordinate of the process that should receive the result. If <code>rdest</code> or <code>cdest = -1</code> , all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value maximization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

?gamn2d

Performs element-wise absolute value minimization.

Syntax

```
call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.

<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be compared with to produce the minimum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> × <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value minimization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

Examples of BLACS Routines Usage

?gsum2d

Performs element-wise summation.

Syntax

```
call igsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call sgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call dgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call cgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call zgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be added to produce the sum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
----------	--

Description

This routine performs element-wise summation, that is, each element of matrix A is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

BLACS Point To Point Communication

This section describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The *send* operation produces a message that is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the *send* or *receive* operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the *send* or *receive* operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```
IAM = MY_PROCESS_ID()
IF (IAM .EQ. 0) THEN
  SEND TO PROCESS 1
  RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
```

```
SEND TO PROCESS 0
  RECV FROM PROCESS 0
END IF
```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them *A*, *B*, and *C*) to process 1, process 1 must receive *A* before it can receive *B*, and message *C* can be received only after both *A* and *B*. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . . , 3 send messages *A*, . . . , *D* to process 4, process 4 may receive these messages in any order that is convenient.

Convention

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

<code>i</code>	integer
<code>s</code>	single precision real
<code>d</code>	double precision real
<code>c</code>	single precision complex
<code>z</code>	double precision complex
<code>ge</code>	The data to be communicated is stored in a general rectangular matrix.
<code>tr</code>	The data to be communicated is stored in a trapezoidal matrix.
<code>sd</code>	Send. One process sends to another.
<code>rv</code>	Receive. One process receives from another.

BLACS Point To Point Communication

Routine name	Operation performed
gesd2d	Take the indicated matrix and send it to the destination process.
trsd2d	
gerv2d	Receive a message from the process into the matrix.
trrv2d	

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector X , which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)

IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
  CALL DGEDSD2D(5, 1, X, 5, 1, 0)
  CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
  CALL DGEDSD2D(5, 1, X, 5, 0, 0)
  CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

?gesd2d

Takes a general rectangular matrix and sends it to the destination process.

Syntax

```
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

See Also

[Examples of BLACS Routines Usage](#)

?trsd2d

Takes a trapezoidal matrix and sends it to the destination process.

Syntax

```
call itrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

```
call dtrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d

Receives a message from the process into the general rectangular matrix.

Syntax

```
call igerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call sgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call dgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call cgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call zgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the source of the message.
<i>csrc</i>	INTEGER. The process column coordinate of the source of the message.

Output Parameters

<i>a</i>	An array of dimension (lda, n) to receive the incoming message into.
----------	--

Description

This routine receives a message from process {RSRC, CSRC} into the general rectangular matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

See Also

[Examples of BLACS Routines Usage](#)

?trrv2d

Receives a message from the process into the trapezoidal matrix.

Syntax

```
call itrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call strrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the source of the message.
<i>csrc</i>	INTEGER. The process column coordinate of the source of the message.

Output Parameters

<i>a</i>	An array of dimension (lda, n) to receive the incoming message into.
----------	--

Description

This routine receives a message from process {RSRC, CSRC} into the trapezoidal matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

BLACS Broadcast Routines

This section describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast/send*. All processes within the same scope must then issue the complementary *broadcast/receive*.

The BLACS define that both broadcast/send and broadcast/receive are *globally-blocking*. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be *locally-blocking*. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with `SCOPE = 'ALL'` must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
bs	Broadcast/send. A process begins the broadcast of data within a scope.
br	Broadcast/receive A process receives and participates in the broadcast of data within a scope.

BLACS Broadcast Routines

Routine name	Operation performed
gebs2d	Start a broadcast along a scope.
trbs2d	
gebr2d	Receive and participate in a broadcast along a scope.
trbr2d	

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebs2d( icontxt, scope, top, m, n, a, lda )
call sgebs2d( icontxt, scope, top, m, n, a, lda )
call dgebs2d( icontxt, scope, top, m, n, a, lda )
call cgebs2d( icontxt, scope, top, m, n, a, lda )
call zgebs2d( icontxt, scope, top, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call strbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

```
call ztrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.
<i>csrc</i>	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix A. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbr2d

Receives and participates in a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call strbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.
<i>csrc</i>	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix A. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

blacs_setup

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```

Input Parameters

nprocs INTEGER. On the process spawned from the keyboard rather than from `pvmspawn`, this parameter indicates the number of processes to create when building the virtual machine.

Output Parameters

mypnum INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.

nprocs INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to `blacs_pinfo`. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

`blacs_setup` is used to allocate the virtual machine and spawn off processes. It reads in a file called `blacs_setup.dat`, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (`PvmTaskDefault`), 4 (`PvmTaskDebug`), 8 (`PvmTaskTrace`), and 12 (`PvmTaskDebug + PvmTaskTrace`). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to `blacs_pinfo` informs you that the virtual machine has not been set up, and a call to `blacs_setup` then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file `blacs_setup.dat` does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

See Also

[Examples of BLACS Routines Usage](#)

blacs_get

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( icontxt, what, val )
```

Input Parameters

icontxt INTEGER. On values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

what INTEGER. Indicates what BLACS internal(s) should be returned in *val*. Present options are:

- *what* = 0 : Handle indicating default system context.
- *what* = 1 : The BLACS message ID range.
- *what* = 2 : The BLACS debug level the library was compiled with.
- *what* = 10 : Handle indicating the system context used to define the BLACS context whose handle is *icontxt*.
- *what* = 11 : Number of rings multiring broadcast topology is presently using.
- *what* = 12 : Number of branches general tree broadcast topology is presently using.
- *what* = 13 : Number of rings multiring combine topology is presently using.
- *what* = 14 : Number of branches general tree combine topology is presently using.
- *what* = 15 : Whether topologies are forced to be repeatable or not. A non-zero return value indicates that topologies are being forced to be repeatable. See [Repeatability and Coherence](#) for more information about repeatability.
- *what* = 16 : Whether topologies are forced to be heterogenous coherent or not. A non-zero return value indicates that topologies are being forced to be heterogenous coherent. See [Repeatability and Coherence](#) for more information about coherence.

Output Parameters

val INTEGER. The value of the BLACS internal.

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into [blacs_gridinit](#) or [blacs_gridmap](#).

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use [blacs_get](#) to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter *icontxt* is unused.

[blacs_get](#) returns information on three quantities that are tied to an individual BLACS context, which is passed in as *icontxt*. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for TOP = 'M' (multiring broadcast/combine)
- The number of branches for TOP = 'T' (general tree broadcast/general tree gather).
- Whether topologies are being forced to be repeatable or heterogenous coherent.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_set](#)

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( icontxt, what, val )
```

Input Parameters

<i>icontxt</i>	INTEGER. For values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	<p>INTEGER. Indicates what BLACS internal(s) should be set. Present values are:</p> <ul style="list-style-type: none"> • 1 = Set the BLACS message ID range • 11 = Number of rings for multiring broadcast topology to use • 12 = Number of branches for general tree broadcast topology to use • 13 = Number of rings for multiring combine topology to use • 14 = Number of branches for general tree combine topology to use • 15 = Force topologies to be repeatable or not • 16 = Force topologies to be heterogenous coherent or not
<i>val</i>	INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on <i>what</i> values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on *what* values:

<i>what</i> = 1	<p>Setting the BLACS message ID range.</p> <p>If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to <code>blacs_gridinit</code> or <code>blacs_gridmap</code>. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter <i>icontxt</i> is ignored, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (2)</p> <p>VAL(1) : The smallest message ID (also called message type or message tag) the BLACS should use.</p> <p>VAL(2) : The largest message ID (also called message type or message tag) the BLACS should use.</p>
<i>what</i> = 11	<p>Set number of rings for TOP = 'M' (multiring broadcast). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) : The number of rings for multiring topology to use.</p>
<i>what</i> = 12	<p>Set number of branches for TOP = 'T' (general tree broadcast). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) : The number of branches for general tree topology to use.</p>

<i>what</i> = 13	<p>Set number of rings for TOP = 'M' (multiring combine). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) : The number of rings for multiring topology to use.</p>
<i>what</i> = 14	<p>Set number of branches for TOP = 'T' (general tree gather). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) : The number of branches for general tree topology to use.</p>
<i>what</i> = 15	<p>Force topologies to be repeatable or not (see Repeatability and Coherence for more information about repeatability).</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) = 0 (default) Topologies are not required to be repeatable.</p> <p>VAL(1) ≠ 0 All used topologies are required to be repeatable, which might degrade performance.</p>
<i>what</i> = 16	<p>Force topologies to be heterogenous coherent or not (see Repeatability and Coherence for more information about coherence).</p> <p>VAL (input) INTEGER array of dimension (1)</p> <p>VAL(1) = 0 (default) Topologies are not required to be heterogenous coherent.</p> <p>VAL(1) ≠ 0 All used topologies are required to be heterogenous coherent, which might degrade performance.</p>

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

call blacs_gridinit(*icontxt*, *layout*, *nprow*, *npcol*)

Input Parameters

<i>icontxt</i>	<p>INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call <code>blacs_get</code> to obtain a default system context.</p>
<i>layout</i>	<p>CHARACTER*1. Indicates how to map processes to BLACS grid. Options are:</p> <ul style="list-style-type: none"> • 'R' : Use row-major natural ordering • 'C' : Use column-major natural ordering • ELSE : Use row-major natural ordering
<i>nprow</i>	<p>INTEGER. Indicates how many process rows the process grid should contain.</p>

npcol INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple $n_{prow} \times n_{pcol}$ process grid. This process grid uses the first $n_{prow} * n_{pcol}$ processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_get](#)

[blacs_gridmap](#)

[blacs_setup](#)

blacs_gridmap

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( icontxt, usermap, ldumap, npro, npcol )
```

Input Parameters

icontxt INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

usermap INTEGER. Array, dimension $(ldumap, npcold)$, indicating the process-to-grid mapping.

ldumap INTEGER. Leading dimension of the 2D array *usermap*. $ldumap \geq npro$.

npro INTEGER. Indicates how many process rows the process grid should contain.

`npcol` INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

`icontxt` INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i, j)` holds the process number of the process to be placed in $\{i, j\}$ of the process grid. On most distributed systems, this process number is a machine defined number between $0 \dots nprow-1$. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a `gridmap` where the first $nprow * npcold$ processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for *multigridding*: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_get](#)

[blacs_gridinit](#)

[blacs_setup](#)

Destruction Routines

This section describes BLACS routines that destroy grids, abort processes, and free resources.

BLACS Destruction Routines

Routine name	Operation performed
<code>blacs_freebuff</code>	Frees BLACS buffer.
<code>blacs_gridexit</code>	Frees a BLACS context.

Routine name	Operation performed
<code>blacs_abort</code>	Aborts all processes.
<code>blacs_exit</code>	Frees all BLACS contexts and releases all allocated memory.

blacs_freebuff

Frees BLACS buffer.

Syntax

```
call blacs_freebuff( icontxt, wait )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context.
<i>wait</i>	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

blacs_gridexit

Frees a BLACS context.

Syntax

```
call blacs_gridexit( icontxt )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context to be freed.
----------------	---

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort

Aborts all processes.

Syntax

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context to be aborted.
<i>errornum</i>	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

`blacs_exit`

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

<i>continue</i>	INTEGER. Flag indicating whether message passing continues after the BLACS are done. If <i>continue</i> is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.
-----------------	---

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM `send/recvs` will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

See Also

[Examples of BLACS Routines Usage](#)

Informational Routines

This section describes BLACS routines that return information involving the process grid.

BLACS Informational Routines

Routine name	Operation performed
blacs_gridinfo	Returns information on the current grid.
blacs_pnum	Returns the system process number of the process in the process grid.
blacs_pcoord	Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( icontxt, pnum, prow, pcol )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

pnum INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a `tid` for PVM.

Output Parameters

prow INTEGER. Row coordinates of the *pnum* process in the BLACS grid.

pcol INTEGER. Column coordinates of the *pnum* process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

See Also

Examples of BLACS Routines Usage

Miscellaneous Routines

This section describes `blacs_barrier` routine.

BLACS Informational Routines

Routine name	Operation performed
<code>blacs_barrier</code>	Holds up execution of all processes within the indicated scope until they have all called the routine.

`blacs_barrier`

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

scope CHARACTER*1. Parameter that indicates whether a process row (*scope*='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Example. BLACS Usage. Hello World

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid.

```

PROGRAM HELLO
*   -- BLACS example code --
*   Written by Clint Whaley 7/26/94
*   Performs a simple check-in type hello world
*
*   ..
*   .. External Functions ..
INTEGER BLACS_PNUM
EXTERNAL BLACS_PNUM
*
*   ..
*   .. Variable Declaration ..
INTEGER CONTXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
INTEGER ICALLER, I, J, HISROW, HISCOL
*
*   Determine my process number and the number of processes in
*   machine
*
CALL BLACS_PINFO(IAM, NPROCS)
*
*   If in PVM, create virtual machine if it doesn't exist
*
IF (NPROCS .LT. 1) THEN
  IF (IAM .EQ. 0) THEN
    WRITE(*, 1000)
    READ(*, 2000) NPROCS
  END IF
  CALL BLACS_SETUP(IAM, NPROCS)
END IF
*
*   Set up process grid that is as close to square as possible
*
NPROW = INT( SQRT( REAL(NPROCS) ) )
NPCOL = NPROCS / NPROW
*
*   Get default system context, and define grid
*
CALL BLACS_GET(0, 0, CONTXT)
CALL BLACS_GRIDINIT(CONTXT, 'Row', NPROW, NPCOL)
CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*   If I'm not in grid, go to end of program
*
IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30
*
*   Get my process ID from my grid coordinates
*
ICALLER = BLACS_PNUM(CONTXT, MYPROW, MYPCOL)
*
*   If I am process {0,0}, receive check-in messages from
*   all nodes
*
IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

```

```

WRITE(*,*) ' '

DO 20 I = 0, NPROW-1
  DO 10 J = 0, NPCOL-1

    IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
      CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
    END IF
*
*   Make sure ICALLER is where we think in process grid
*
    CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
    IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
      WRITE(*,*) 'Grid error! Halting . . .'

      STOP
    END IF
    WRITE(*, 3000) I, J, ICALLER

10    CONTINUE
20    CONTINUE
    WRITE(*,*) ' '
    WRITE(*,*) 'All processes checked in. Run finished.'
*
*   All processes but {0,0} send process ID as a check-in
*
    ELSE

      CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)
    END IF

30    CONTINUE

    CALL BLACS_EXIT(0)

1000  FORMAT('How many processes in machine?')
2000  FORMAT(I)
3000  FORMAT('Process {',i2,',',i2,'} (node number =',I,
$      ') has checked in.')

    STOP
  END

```

Example. BLACS Usage. PROCMAP

This routine maps processes to a grid using `blacs_gridmap`.

```

SUBROUTINE PROCMAP(CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL, IMAP)
*
*   -- BLACS example code --
*
*   Written by Clint Whaley 7/26/94

```

```

*      ..
*      .. Scalar Arguments ..
*      INTEGER CONTEXT, MAPPING, BEGPCROC, NPROW, NPCOL
*
*      ..
*      .. Array Arguments ..
*      INTEGER IMAP(NPROW, *)
*      ..
*
* Purpose
* =====
* PROCMAP maps NPROW*NPCOL processes starting from process BEGPCROC to
* the grid in a variety of ways depending on the parameter MAPPING.
*
* Arguments
* =====
*
* CONTEXT      (output) INTEGER
*               This integer is used by the BLACS to indicate a context.
*               A context is a universe where messages exist and do not
*               interact with other context's messages. The context
*               includes the definition of a grid, and each process's
*               coordinates in it.
*
* MAPPING      (input) INTEGER
*               Way to map processes to grid. Choices are:
*               1 : row-major natural ordering
*               2 : column-major natural ordering
*
* BEGPCROC     (input) INTEGER
*               The process number (between 0 and NPROCS-1) to use as
*
*               {0,0}. From this process, processes will be assigned
*               to the grid as indicated by MAPPING.
*
* NPROW        (input) INTEGER
*               The number of process rows the created grid
*
*               should have.
*
* NPCOL        (input) INTEGER
*               The number of process columns the created grid
*
*               should have.
*
* IMAP         (workspace) INTEGER array of dimension (NPROW, NPCOL)
*               Workspace, where the array which maps the
*
*               processes to the grid will be stored for the
*               call to GRIDMAP.
*
* =====
*
*      ..
*      .. External Functions ..
*      INTEGER BLACS_PNUM
*
*      EXTERNAL BLACS_PNUM

```

```

*      ..
*      .. External Subroutines ..
EXTERNAL BLACS_PINFO, BLACS_GRIDINIT, BLACS_GRIDMAP
*      ..
*      .. Local Scalars ..
INTEGER TMPCONXT, NPROCS, I, J, K
*
*      ..
*      .. Executable Statements ..
*
*      See how many processes there are in the system
*
CALL BLACS_PINFO( I, NPROCS )

IF (NPROCS-BEGPROC .LT. NPROW*NPCOL) THEN
  WRITE(*,*) 'Not enough processes for grid'
  STOP
END IF
*
*      Temporarily map all processes into 1 x NPROCS grid
*
*
CALL BLACS_GET( 0, 0, TMPCONXT )
CALL BLACS_GRIDINIT( TMPCONXT, 'Row', 1, NPROCS )
K = BEGPROC
*
*      If we want a row-major natural ordering
*
*
IF (MAPPING .EQ. 1) THEN

  DO I = 1, NPROW
    DO J = 1, NPCOL
      IMAP(I, J) = BLACS_PNUM(TMPCONXT, 0, K)
      K = K + 1W
    END DO
  END DO
*
*      If we want a column-major natural ordering
*
*
ELSE IF (MAPPING .EQ. 2) THEN

  DO J = 1, NPCOL
    DO I = 1, NPROW
      IMAP(I, J) = BLACS_PNUM(TMPCONXT, 0, K)

      K = K + 1

    END DO
  END DO
ELSE
  WRITE(*,*) 'Unknown mapping.'

```

```

        STOP
    END IF

*
*   Free temporary context
*
    CALL BLACS_GRIDEXIT(TMPCONXT)
*
*   Apply the new mapping to form desired context
*
    CALL BLACS_GET( 0, 0, CONTEXT )
    CALL BLACS_GRIDMAP( CONTEXT, IMAP, NPROW, NPROW, NPCOL )

*
*
*
    RETURN
END

```

Example. BLACS Usage. PARALLEL DOT PRODUCT

This routine does a bone-headed parallel double precision dot product of two vectors. Arguments are input on process {0,0}, and output everywhere else.

```

    DOUBLE PRECISION FUNCTION PDDOT( CONTEXT, N, X, Y )
*
*   -- BLACS example code --
*
*   Written by Clint Whaley 7/26/94
*   ..
*   .. Scalar Arguments ..
    INTEGER CONTEXT, N
*   ..
*   .. Array Arguments ..
    DOUBLE PRECISION X(*), Y(*)
*   ..
*
*   Purpose
*   =====
*   PDDOT is a restricted parallel version of the BLAS routine
*   DDOT. It assumes that the increment on both vectors is one,
*   and that process {0,0} starts out owning the vectors and
*
*   has N. It returns the dot product of the two N-length vectors
*   X and Y, that is, PDDOT = X' Y.
*
*   Arguments
*
*   =====
*
*   CONTEXT      (input) INTEGER
*                 This integer is used by the BLACS to indicate a context.
*                 A context is a universe where messages exist and do not
*                 interact with other context's messages. The context
*                 includes the definition of a grid, and each process's
*                 coordinates in it.
*
*   N            (input/output) INTEGER
*                 The length of the vectors X and Y. Input
*                 for {0,0}, output for everyone else.

```

```

*
* X          (input/output) DOUBLE PRECISION array of dimension (N)
*           The vector X of PDDOT = X' Y. Input for {0,0},
*           output for everyone else.
*
* Y          (input/output) DOUBLE PRECISION array of dimension (N)
*           The vector Y of PDDOT = X' Y. Input for {0,0},
*           output for everyone else.
*
* =====
*
* ..
* .. External Functions ..
* DOUBLE PRECISION DDOT
*
* EXTERNAL DDOT
*
* ..
* .. External Subroutines ..
* EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D
*
* ..
* .. Local Scalars ..
* INTEGER IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL, I, LN
*
* DOUBLE PRECISION LDDOT
*
* ..
* .. Executable Statements ..
*
* Find out what grid has been set up, and pretend it is 1-D
*
* CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )
*
* IAM = MYPROW*NPCOL + MYPCOL
* NPROCS = NPROW * NPCOL
*
* Temporarily map all processes into 1 x NPROCS grid
*
* CALL BLACS_GET( 0, 0, TMPCONTXT )
* CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
* K = BEGPROC
*
* Do bone-headed thing, and just send entire X and Y to
*
* everyone
*
* IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN
*
*   CALL IGEBS2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1 )
*
*   CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, X, N )

```

```

    CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, Y, N )
ELSE
    CALL IGEBR2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, X, N, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, Y, N, 0, 0 )
ENDIF
*
* Find out the number of local rows to multiply (LN), and
*
* where in vectors to start (I)
*
*
LN = N / NPROCS
I = 1 + IAM * LN
*
* Last process does any extra rows
*
IF (IAM .EQ. NPROCS-1) LN = LN + MOD(N, NPROCS)
*
* Figure dot product of my piece of X and Y
*
LDDOT = DDOT( LN, X(I), 1, Y(I), 1 )
*
* Add local dot products to get global dot product;
*
* give all procs the answer
*
CALL DGSUM2D( CONTXT, 'All', '1-tree', 1, 1, LDDOT, 1, -1, 0 )

PDDOT = LDDOT

RETURN

END

```

Example. BLACS Usage. PARALLEL MATRIX INFINITY NORM

This routine does a parallel infinity norm on a distributed double precision matrix. Unlike the PDDOT example, this routine assumes the matrix has already been distributed.

```

DOUBLE PRECISION FUNCTION PDINFNRM(CONTXT, LM, LN, A, LDA, WORK)
*
* -- BLACS example code --
*
* Written by Clint Whaley.
* ..
* .. Scalar Arguments ..
INTEGER CONTEXT, LM, LN, LDA
*
* ..
* .. Array Arguments ..

```

```

      DOUBLE PRECISION A(LDA, *), WORK(*)
*
*   ..
*
* Purpose
* =====
* Compute the infinity norm of a distributed matrix, where
* the matrix is spread across a 2D process grid. The result is
* left on all processes.
*
* Arguments
* =====
*
* CONTEXT      (input) INTEGER
*              This integer is used by the BLACS to indicate a context.
*              A context is a universe where messages exist and do not
*              interact with other context's messages. The context
*              includes the definition of a grid, and each process's
*              coordinates in it.
*
* LM           (input) INTEGER
*              Number of rows of the global matrix owned by this
*              process.
*
* LN           (input) INTEGER
*              Number of columns of the global matrix owned by this
*              process.
*
* A            (input) DOUBLE PRECISION, dimension (LDA,N)
*              The matrix whose norm you wish to compute.
*
* LDA         (input) INTEGER
*              Leading Dimension of A.
*
* WORK        (temporary) DOUBLE PRECISION array, dimension (LM)
*              Temporary work space used for summing rows.
*
*
* .. External Subroutines ..
* EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D, DGAMX2D
*
* ..
* .. External Functions ..
* INTEGER IDAMAX
* DOUBLE PRECISION DASUM
*
* .. Local Scalars ..
* INTEGER NPROW, NPCOL, MYROW, MYCOL, I, J
*
* DOUBLE PRECISION MAX
*
* .. Executable Statements ..

```

```
*
*   Get process grid information
*
*   CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPCOL )
*
*   Add all local rows together
*
*
*   DO 20 I = 1, LM
*
*       WORK(I) = DASUM(LN, A(I,1), LDA)
20  CONTINUE
*
*   Find sum of global matrix rows and store on column 0 of
*
*   process grid
*
*
*   CALL DGSUM2D(CONTXT, 'Row', '1-tree', LM, 1, WORK, LM, MYROW, 0)
*
*   Find maximum sum of rows for supnorm
*
*
*   IF (MYCOL .EQ. 0) THEN
*
*       MAX = WORK(IDAMAX(LM,WORK,1))
*
*       IF (LM .LT. 1) MAX = 0.0D0
*
*       CALL DGAMX2D(CONTXT, 'Col', 'h', 1, 1, MAX, 1, I, I, -1, -1, 0)
*   END IF
*
*   Process column 0 has answer; send answer to all nodes
*
*
*   IF (MYCOL .EQ. 0) THEN
*
*       CALL DGEBS2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1)
*
*   ELSE
*
*       CALL DGEBR2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1, 0, 0)
*
*   END IF
*
*   PDINFNRM = MAX
```

```
*  
  RETURN  
*  
  End of PDINFNR  
*  
  END
```


Data Fitting Functions

Data Fitting functions in Intel® MKL provide spline-based interpolation capabilities that you can use to approximate functions, function derivatives or integrals, and perform cell search operations.

The Data Fitting component is task based. The task is a data structure or descriptor that holds the parameters related to a specific Data Fitting operation. You can modify the task parameters using the task editing functionality of the library.

For definition of the implemented operations, see [Mathematical Conventions](#).

Data Fitting routines use the following workflow to process a task:

1. Create a task or multiple tasks.
2. Modify the task parameters.
3. Perform a Data Fitting computation.
4. Destroy the task or tasks.

All Data Fitting functions fall into the following categories:

[Task Creation and Initialization Routines](#) - routines that create a new Data Fitting task descriptor and initialize the most common parameters, such as partition of the interpolation interval, values of the vector-valued function, and the parameters describing their structure.

[Task Configuration Routines](#) - routines that set, modify, or query parameters in an existing Data Fitting task.

[Computational Routines](#) - routines that perform Data Fitting computations, such as construction of a spline, interpolation, computation of derivatives and integrals, and search.

[Task Destructors](#) - routines that delete Data Fitting task descriptors and deallocate resources.

You can access the Data Fitting routines through the Fortran and C89/C99 language interfaces. You can also use the C89 interface with more recent versions of C/C++, or the Fortran 90 interface with programs written in Fortran 95.

The `$(MKL)/include` directory of the Intel® MKL contains the following Data Fitting header files:

- `mkl_df.f90`

You can find examples that demonstrate usage of Data Fitting routines in the `$(MKL)/examples/datafittingf` directory.

Naming Conventions

The interfaces of the Data Fitting functions are in lowercase, while the names of the types and constants are in uppercase.

The names of all routines have the following structure:

```
df[datatype]<base_name>
```

where

- `df` is a prefix indicating that the routine belongs to the Data Fitting component of Intel MKL.
- `[datatype]` field specifies the type of the input and/or output data and can be `s` (for the single precision real type), `d` (for the double precision real type), or `i` (for the integer type). This field is omitted in the names of the routines that are not data type dependent.
- `<base_name>` field specifies the functionality the routine performs. For example, this field can be `newtask1d`, `interpolate1d`, or `deletetask`

Data Types

The Data Fitting component provides routines for processing single and double precision real data types. The results of cell search operations are returned as a generic integer data type.

All Data Fitting routines use the following data type:

Type	Data Object
TYPE (DF_TASK)	Pointer to a task

NOTE

The actual size of the generic integer type is platform-dependent. Before compiling your application, you need to set an appropriate byte size for integers. For details, see section *Using the ILP64 Interface vs. LP64 Interface* of the Intel® MKL User's Guide.

Mathematical Conventions for Data Fitting Functions

This section explains the notation used for Data Fitting function descriptions. Spline notations are based on the terminology and definitions of [deBoor2001]. The Subbotin quadratic spline definition follows the conventions of [StechSub76]. The quasi-uniform partition definition is based on [Schumaker2007].

Mathematical Notation in the Data Fitting Component

Concept	Mathematical Notation
Partition of interpolation interval $[a, b]$, where <ul style="list-style-type: none"> x_i denotes breakpoints. $[x_i, x_{i+1})$ denotes a sub-interval (cell) of size $\Delta_i = x_{i+1} - x_i$. 	$\{x_i\}_{i=1, \dots, n}$, where $a = x_1 < x_2 < \dots < x_n = b$
Quasi-uniform partition of interpolation interval $[a, b]$	Partition $\{x_i\}_{i=1, \dots, n}$ which meets the constraint with a constant C defined as $1 \leq M / m \leq C,$ where <ul style="list-style-type: none"> $M = \max_{i=1, \dots, n-1} (\Delta_i)$ $m = \min_{i=1, \dots, n-1} (\Delta_i)$ $\Delta_i = x_{i+1} - x_i$
Vector-valued function of dimension p being fit	$f(x) = (f_1(x), \dots, f_p(x))$
Piecewise polynomial (PP) function f of order $k+1$	$f(x) = P_i(x)$, if $x \in [x_i, x_{i+1})$, $i = 1, \dots, n-1$ where <ul style="list-style-type: none"> $\{x_i\}_{i=1, \dots, n}$ is a strictly increasing sequence of breakpoints. $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$ is a polynomial of degree k (order $k+1$) over the interval $x \in [x_i, x_{i+1})$.
Function p agrees with function f at the points $\{x_i\}_{i=1, \dots, n}$.	For every point ζ in sequence $\{x_i\}_{i=1, \dots, n}$ that occurs m times, the equality $p^{(i-1)}(\zeta) = f^{(i-1)}(\zeta)$ holds for all $i = 1, \dots, m$, where $p^{(i)}(t)$ is the derivative of the i -th order.

Concept	Mathematical Notation
The k -th divided difference of function f at points x_i, \dots, x_{i+k} . This difference is the leading coefficient of the polynomial of order $k+1$ that agrees with f at x_i, \dots, x_{i+k} .	$[x_i, \dots, x_{i+k}] f$ In particular, <ul style="list-style-type: none"> $[x_1] f = f(x_1)$ $[x_1, x_2] f = (f(x_1) - f(x_2)) / (x_1 - x_2)$
A k -order derivative of interpolant $f(x)$ at interpolation site τ .	$f^{(k)}(\tau)$

Interpolants to the Function f at x_1, \dots, x_n and Boundary Conditions

Concept	Mathematical Notation
Linear interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i)$, where <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = [x_i, x_{i+1}] f$ $i = 1, \dots, n-1$
Piecewise parabolic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2$, $x \in [x_i, x_{i+1})$ Coefficients $c_{1,i}$, $c_{2,i}$, and $c_{3,i}$ depend on the conditions: <ul style="list-style-type: none"> $P_i(x_i) = f(x_i)$ $P_i(x_{i+1}) = f(x_{i+1})$ $P_i((x_{i+1} + x_i) / 2) = v_{i+1}$ where parameter v_{i+1} depends on the interpolant being continuously differentiable: $P_{i-1}^{(1)}(x_i) = P_i^{(1)}(x_i)$
Piecewise parabolic Subbotin interpolant	$P(x) = P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + d_{3,i}((x - t_i)_+)^2$, where <ul style="list-style-type: none"> $x \in [t_i, t_{i+1})$ $\{t_j\}_{j=1, \dots, n+1}$ is a sequence of knots such that <ul style="list-style-type: none"> $t_1 = x_1, t_{n+1} = x_n$ $t_i \in (x_{i-1}, x_i)$, $i = 2, \dots, n$ $x_+ = f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$ Coefficients $c_{1,i}$, $c_{2,i}$, $c_{3,i}$, and $d_{3,i}$ depend on the following conditions: <ul style="list-style-type: none"> $P_i(x_i) = f(x_i)$, $P_i(x_{i+1}) = f(x_{i+1})$ $P(x)$ is a continuously differentiable polynomial of the second degree on $[t_i, t_{i+1})$, $i = 1, \dots, n$.
Piecewise cubic Hermite interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$, where <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = s_i$ $c_{3,i} = ([x_i, x_{i+1}] f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}] f) / (\Delta x_i)^2$

Concept	Mathematical Notation
Piecewise cubic Bessel interpolant	<ul style="list-style-type: none"> • $i = 1, \dots, n-1$ • $s_i = f^{(1)}(x_i)$ $P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • $s_i = (\Delta x_i[x_{i-1}, x_i]f + \Delta x_{i-1}[x_i, x_{i+1}]f) / (\Delta x_i + \Delta x_{i+1})$
Piecewise cubic Akima interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • $s_i = (w_{i+1}[x_{i-1}, x_i]f + w_{i-1}[x_i, x_{i+1}]f) / (w_{i+1} + w_{i-1}),$ <p>where</p> $w_i = [x_i, x_{i+1}]f - [x_{i-1}, x_i]f $
Piecewise natural cubic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • Parameter s_i depends on the condition that the interpolant is twice continuously differentiable: $P_{i-1}^{(2)}(x_i) = P_i^{(2)}(x_i)$.
Not-a-knot boundary condition.	Parameters s_1 and s_n provide $P_1 = P_2$ and $P_{n-1} = P_n$, so that the first and the last interior breakpoints are inactive.
Free-end boundary condition.	$f''(x_1) = f''(x_n) = 0$
Look-up interpolator for discrete set of points $(x_1, y_1), \dots, (x_n, y_n)$.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x = x_2 \\ \dots & \dots \\ y_n, & \text{if } x = x_n \\ \text{error,} & \text{otherwise} \end{cases}$

Concept	Mathematical Notation
Step-wise constant continuous right interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x_1 \leq x < x_2 \\ y_2, & \text{if } x_2 \leq x < x_3 \\ \dots & \\ y_{n-1}, & \text{if } x_{n-1} \leq x < x_n \\ y_n, & \text{if } x = x_n \end{cases}$
Step-wise constant continuous left interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x_1 < x \leq x_2 \\ y_3, & \text{if } x_2 < x \leq x_3 \\ \dots & \\ y_n, & \text{if } x_{n-1} < x \leq x_n \end{cases}$

Data Fitting Usage Model

Consider an algorithm that uses the Data Fitting functions. Typically, such algorithms consist of four steps or stages:

1. Create a task. You can call the Data Fitting function several times to create multiple tasks.

```
status = dfdnewtaskld( task, nx, x, xhint, ny, y, yhint );
```

2. Modify the task parameters.

```
status = dfdeditppspline1d( task, s_order, c_type, bc_type, bc, ic_type, ic,
scoeff, scoeffhint );
```

3. Perform Data Fitting spline-based computations. You may reiterate steps 2-3 as needed.

```
status = dfdinterpolate1d(task, estimate, method, nsite, site, sitehint, ndorder,
dorder, datahint, r, rhint, cell );
```

4. Destroy the task or tasks.

```
status = dfdeletetask( task );
```

See Also

[Data Fitting Usage Examples](#)

Data Fitting Usage Examples

You can get Fortran source code in the `.\examples\datafittingf` subdirectory of the Intel MKL installation directory.

Task Status and Error Reporting

The Data Fitting routines report a task status through integer values. Negative status values indicate errors, while positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The status codes have symbolic names predefined in the header file as integer constants via the `PARAMETER` operators.

If no error occurred, the function returns the `DF_STATUS_OK` code defined as zero:

INTEGER, PARAMETER::DF_STATUS_OK = 0

In case of an error, the function returns a non-zero error code that specifies the origin of the failure. Header files define the following status codes:

Status Codes in the Data Fitting Component

Status Code	Description
Common Status Codes	
DF_STATUS_OK	Operation completed successfully.
DF_ERROR_NULL_TASK	Data Fitting task is a NULL pointer.
DF_ERROR_MEM_FAILURE	Memory allocation failure.
DF_ERROR_METHOD_NOT_SUPPORTED	Requested method is not supported.
DF_ERROR_COMP_TYPE_NOT_SUPPORTED	Requested computation type is not supported.
DF_ERROR_NULL_PTR	Pointer to parameter is null.
Data Fitting Task Creation and Initialization, and Generic Editing Operations	
DF_ERROR_BAD_NX	Invalid number of breakpoints.
DF_ERROR_BAD_X	Array of breakpoints is invalid.
DF_ERROR_BAD_X_HINT	Invalid hint describing the structure of the partition.
DF_ERROR_BAD_NY	Invalid dimension of vector-valued function y .
DF_ERROR_BAD_Y	Array of function values is invalid.
DF_ERROR_BAD_Y_HINT	Invalid flag describing the structure of function y
Data Fitting Task-Specific Editing Operations	
DF_ERROR_BAD_SPLINE_ORDER	Invalid spline order.
DF_ERROR_BAD_SPLINE_TYPE	Invalid spline type.
DF_ERROR_BAD_IC_TYPE	Type of internal conditions used for spline construction is invalid.
DF_ERROR_BAD_IC	Array of internal conditions for spline construction is not defined.
DF_ERROR_BAD_BC_TYPE	Type of boundary conditions used in spline construction is invalid.
DF_ERROR_BAD_BC	Array of boundary conditions for spline construction is not defined.
DF_ERROR_BAD_PP_COEFF	Array of piecewise polynomial spline coefficients is not defined.
DF_ERROR_BAD_PP_COEFF_HINT	Invalid flag describing the structure of the piecewise polynomial spline coefficients.
DF_ERROR_BAD_PERIODIC_VAL	Function values at the endpoints of the interpolation interval are not equal as required in periodic boundary conditions.

Status Code	Description
DF_ERROR_BAD_DATA_ATTR	Invalid attribute of the pointer to be set or modified in Data Fitting task descriptor with the <code>df?editidxptr</code> task editor.
DF_ERROR_BAD_DATA_IDX	Index of the pointer to be set or modified in the Data Fitting task descriptor with the <code>df?editidxptr</code> task editor is out of the pre-defined range.
Data Fitting Computation Operations	
DF_ERROR_BAD_NSITE	Invalid number of interpolation sites.
DF_ERROR_BAD_SITE	Array of interpolation sites is not defined.
DF_ERROR_BAD_SITE_HINT	Invalid flag describing the structure of interpolation sites.
DF_ERROR_BAD_NDORDER	Invalid size of the array defining derivative orders to be computed at interpolation sites.
DF_ERROR_BAD_DORDER	Array defining derivative orders to be computed at interpolation sites is not defined.
DF_ERROR_BAD_DATA_HINT	Invalid flag providing additional information about partition or interpolation sites.
DF_ERROR_BAD_INTERP	Array of spline-based interpolation results is not defined.
DF_ERROR_BAD_INTERP_HINT	Invalid flag defining the structure of spline-based interpolation results.
DF_ERROR_BAD_CELL_IDX	Array of indices of partition cells containing interpolation sites is not defined.
DF_ERROR_BAD_NLIM	Invalid size of arrays containing integration limits.
DF_ERROR_BAD_LLM	Array of the left-side integration limits is not defined.
DF_ERROR_BAD_RLM	Array of the right-side integration limits is not defined.
DF_ERROR_BAD_INTEGR	Array of spline-based integration results is not defined.
DF_ERROR_BAD_INTEGR_HINT	Invalid flag providing the structure of the array of spline-based integration results.
DF_ERROR_BAD_LOOKUP_INTERP_SITE	Bad site provided for interpolation with look-up interpolator.

NOTE

The routine that estimates piecewise polynomial cubic spline coefficients can return internal error codes related to the specifics of the implementation. Such error codes indicate invalid input data or other issues unrelated to Data Fitting routines.

Task Creation and Initialization Routines

Task creation and initialization routines are functions used to create a new task descriptor and initialize its parameters. The Data Fitting component provides the `df?newtaskld` routine that creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

`df?newtaskld`

Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

Syntax

```
status = dfsnewtaskld(task, nx, x, xhint, ny, y, yhint)
```

```
status = dfdnewtaskld(task, nx, x, xhint, ny, y, yhint)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>nx</code>	INTEGER	Number of breakpoints representing partition of interpolation interval $[a, b]$.
<code>x</code>	REAL(KIND=4) DIMENSION(*) for <code>dfsnewtaskld</code> REAL(KIND=8) DIMENSION(*) for <code>dfdnewtaskld</code>	One-dimensional array containing the strictly sorted breakpoints from interpolation interval $[a, b]$. The structure of the array is defined by parameter <code>xhint</code> : <ul style="list-style-type: none"> • If partition is non-uniform or quasi-uniform, the array should contain <code>nx</code> strictly ordered values. • If partition is uniform, the array should contain two entries that represent endpoints of interpolation interval $[a, b]$.
<code>xhint</code>	INTEGER	A flag describing the structure of partition <code>x</code> . For the list of possible values of <code>xhint</code> , see table " Hint Values for Partition x ". If you set the flag to the <code>DF_NO_HINT</code> value, the library interprets the partition as non-uniform.
<code>ny</code>	INTEGER	Dimension of vector-valued function <code>y</code> .
<code>y</code>	REAL(KIND=4) DIMENSION(*) for <code>dfsnewtaskld</code> REAL(KIND=8) DIMENSION(*) for <code>dfdnewtaskld</code>	Vector-valued function <code>y</code> , array of size <code>nx*ny</code> . The storage format of function values in the array is defined by the value of flag <code>yhint</code> .
<code>yhint</code>	INTEGER	A flag describing the structure of array <code>y</code> . Valid hint values are listed in table " Hint Values for Vector-Valued Function y ". If you set the flag to the <code>DF_NO_HINT</code> value, the library assumes that all <code>ny</code> coordinates of the vector-valued function <code>y</code> are provided and stored in row-major format.

Output Parameters

Name	Type	Description
<i>task</i>	TYPE (DF_TASK)	Descriptor of the task.
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the task is created successfully. • Non-zero error code if the task creation failed. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?newtask1d` routine creates and initializes a new Data Fitting task descriptor with user-specified parameters for a one-dimensional Data Fitting task. The x and nx parameters representing the partition of interpolation interval $[a, b]$ are mandatory. If you provide invalid values for these parameters, such as a NULL pointer x or the number of breakpoints smaller than two, the routine does not create the Data Fitting task and returns an error code.

If you provide a vector-valued function y , make sure that the function dimension ny and the array of function values y are both valid. If any of these parameters are invalid, the routine does not create the Data Fitting task and returns an error code.

If you store coordinates of the vector-valued function y in non-contiguous memory locations, you can set the *yhint* flag to `DF_1ST_COORDINATE`, and pass only the first coordinate of the function into the task creation routine. After successful creation of the Data Fitting task, you can pass the remaining coordinates using the `df?editidxptr` task editor.

If the routine fails to create the task descriptor, it returns a NULL task pointer.

The routine supports the following hint values for partition x :

Hint Values for Partition x

Value	Description
<code>DF_NON_UNIFORM_PARTITION</code>	Partition is non-uniform.
<code>DF_QUASI_UNIFORM_PARTITION</code>	Partition is quasi-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition is uniform.
<code>DF_NO_HINT</code>	No hint is provided. By default, partition is interpreted as non-uniform.

The routine supports the following hint values for the vector-valued function:

Hint Values for Vector-Valued Function y

Value	Description
<code>DF_MATRIX_STORAGE_ROWS</code>	Data is stored in row-major format according to C conventions.
<code>DF_MATRIX_STORAGE_COLS</code>	Data is stored in column-major format according to Fortran conventions.
<code>DF_1ST_COORDINATE</code>	The first coordinate of vector-valued data is provided.
<code>DF_NO_HINT</code>	No hint is provided. By default, the coordinates of vector-valued function y are provided and stored in row-major format.

NOTE

You must preserve the arrays x (breakpoints) and y (vector-valued functions) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

Task Configuration Routines

In order to configure tasks, you can use task editors and task query routines.

Task editors initialize or change the predefined Data Fitting task parameters. You can use task editors to initialize or modify pointers to arrays or parameter values.

Task editors can be task-specific or generic. Task-specific editors can modify more than one parameter related to a specific task. Generic editors modify a single parameter at a time.

The Data Fitting component of Intel MKL provides the following task editors:

Data Fitting Task Editors

Editor	Description	Type
<code>df?</code> <code>editppspline1d</code>	Changes parameters of the piecewise polynomial spline.	Task-specific
<code>df?editptr</code>	Changes a pointer in the task descriptor.	Generic
<code>dfieditval</code>	Changes a value in the task descriptor.	Generic
<code>df?editidxptr</code>	Changes a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

Task query routines are used to read the predefined Data Fitting task parameters. You can use task query routines to read the values of pointers or parameters.

Task query routines are generic (not task-specific), allowing you to read a single parameter at a time.

The Data Fitting component of the Intel MKL provides the following task query routines:

Data Fitting Task Query Routines

Editor	Description	Type
<code>df?queryptr</code>	Queries a pointer in the task descriptor.	Generic
<code>dfiqueryval</code>	Queries a value in the task descriptor.	Generic
<code>df?queryidxptr</code>	Queries a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

`df?editppspline1d`

Modifies parameters representing a spline in a Data Fitting task descriptor.

Syntax

```
status = dfseditppspline1d(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff,
scoeffhint)
```

```
status = dfdeditppspline1d(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff,
scoeffhint)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	TYPE (DF_TASK)	Descriptor of the task.
<code>s_order</code>	INTEGER	Spline order. The parameter takes one of the values described in table " Spline Orders Supported by Data Fitting Functions ".
<code>s_type</code>	INTEGER	Spline type. The parameter takes one of the values described in table " Spline Types Supported by Data Fitting Functions ".
<code>bc_type</code>	INTEGER	Type of boundary conditions. The parameter takes one of the values described in table " Boundary Conditions Supported by Data Fitting Functions ".
<code>bc</code>	REAL(KIND=4) DIMENSION(*) for <code>dfseditppspline1d</code> REAL(KIND=8) DIMENSION(*) for <code>dfdeditppspline1d</code>	<p>Pointer to boundary conditions. The size of the array is defined by the value of parameter <code>bc_type</code>:</p> <ul style="list-style-type: none"> • If you set free-end or not-a-knot boundary conditions, pass the <code>NULL</code> pointer to this parameter. • If you combine boundary conditions at the endpoints of the interpolation interval, pass an array of two elements. • If you set a boundary condition for the default quadratic spline or a periodic condition for Hermite or the default cubic spline, pass an array of one element.
<code>ic_type</code>	INTEGER	Type of internal conditions. The parameter takes one of the values described in table " Internal Conditions Supported by Data Fitting Functions ".
<code>ic</code>	REAL(KIND=4) DIMENSION(*) for <code>dfseditppspline1d</code> REAL(KIND=8) DIMENSION(*) for <code>dfdeditppspline1d</code>	<p>A non-NULL pointer to the array of internal conditions. The size of the array is defined by the value of parameter <code>ic_type</code>:</p> <ul style="list-style-type: none"> • If you set <code>first derivatives</code> or <code>second derivatives internal conditions</code> (<code>ic_type=DF_IC_1ST_DER</code> or <code>ic_type=DF_IC_2ND_DER</code>), pass an array of <code>n-1</code> derivative values at the internal points of the interpolation interval. • If you set the <code>knot values</code> internal condition for Subbotin spline (<code>ic_type=DF_IC_Q_KNOT</code>) and the knot partition is non-uniform, pass an array of <code>n+1</code> elements. • If you set the <code>knot values</code> internal condition for Subbotin spline (<code>ic_type=DF_IC_Q_KNOT</code>) and the knot partition is uniform, pass an array of four elements.

Name	Type	Description
<i>scoeff</i>	REAL(KIND=4) DIMENSION(*) for <i>dfseditppspline1d</i> REAL(KIND=8) DIMENSION(*) for <i>dfdeditppspline1d</i>	Spline coefficients. An array of size $ny * s_order * (nx - 1)$. The storage format of the coefficients in the array is defined by the value of flag <i>scoeffhint</i> .
<i>scoeffhint</i>	INTEGER	A flag describing the structure of the array of spline coefficients. For valid hint values, see table " Hint Values for Spline Coefficients ". The library stores the coefficients in row-major format. The default value is <code>DF_NO_HINT</code> .

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> <code>DF_STATUS_OK</code> if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

The editor modifies parameters that describe the order, type, boundary conditions, internal conditions, and coefficients of a spline. The spline order definition is provided in the "[Mathematical Conventions](#)" section. You can set the spline order to any value supported by Data Fitting functions. The table below lists the available values:

Spline Orders Supported by the Data Fitting Functions

Order	Description
<code>DF_PP_STD</code>	Artificial value. Use this value for look-up and step-wise constant interpolants only.
<code>DF_PP_LINEAR</code>	Piecewise polynomial spline of the second order (linear spline).
<code>DF_PP_QUADRATIC</code>	Piecewise polynomial spline of the third order (quadratic spline).
<code>DF_PP_CUBIC</code>	Piecewise polynomial spline of the fourth order (cubic spline).

To perform computations with a spline not supported by Data Fitting routines, set the parameter defining the spline order and pass the spline coefficients to the library in the supported format. For format description, see figure "[Row-major Coefficient Storage Format](#)".

The table below lists the supported spline types:

Spline Types Supported by Data Fitting Functions

Type	Description
<code>DF_PP_DEFAULT</code>	The default spline type. You can use this type with linear, quadratic, or user-defined splines.

Type	Description
DF_PP_SUBBOTIN	Quadratic splines based on Subbotin algorithm, [StechSub76].
DF_PP_NATURAL	Natural cubic spline.
DF_PP_HERMITE	Hermite cubic spline.
DF_PP_BESSEL	Bessel cubic spline.
DF_PP_AKIMA	Akima cubic spline.
DF_LOOKUP_INTERPOLANT	Look-up interpolant.
DF_CR_STEPWISE_CONST_INTERPOLANT	Continuous right step-wise constant interpolant.
DF_CL_STEPWISE_CONST_INTERPOLANT	Continuous left step-wise constant interpolant.

If you perform computations with look-up or step-wise constant interpolants, set the spline order to the `DF_PP_STD` value.

Construction of specific splines may require boundary or internal conditions. To compute coefficients of such splines, you should pass boundary or internal conditions to the library by specifying the type of the conditions and providing the necessary values. For splines that do not require additional conditions, such as linear splines, set condition types to `DF_NO_BC` and `DF_NO_IC`, and pass `NULL` pointers to the conditions. The table below defines the supported boundary conditions:

Boundary Conditions Supported by Data Fitting Functions

Boundary Condition	Description	Spline
DF_NO_BC	No boundary conditions provided.	All
DF_BC_NOT_A_KNOT	Not-a-knot boundary conditions.	Akima, Bessel, Hermite, natural cubic
DF_BC_FREE_END	Free-end boundary conditions.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_LEFT_DER	The first derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_RIGHT_DER	The first derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ST_LEFT_DER	The second derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ND_RIGHT_DER	The second derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_PERIODIC	Periodic boundary conditions.	Linear, all cubic splines
DF_BC_Q_VAL	Function value at point $(x_0 + x_1)/2$	Default quadratic

NOTE

To construct a natural cubic spline, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- `DF_BC_FREE_END` as the boundary condition.

To construct a cubic spline with other boundary conditions, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- the required type of boundary condition.

For Akima, Hermite, Bessel and default cubic splines use the corresponding type defined in [Table Spline Types Supported by Data Fitting Functions](#).

You can combine the values of boundary conditions with a bitwise `OR` operation. This permits you to pass combinations of first and second derivatives at the endpoints of the interpolation interval into the library. To pass a first derivative at the left endpoint and a second derivative at the right endpoint, set the boundary conditions to `DF_BC_1ST_LEFT_DER OR DF_BC_2ND_RIGHT_DER`.

You should pass the combined boundary conditions as an array of two elements. The first entry of the array contains the value of the boundary condition for the left endpoint of the interpolation interval, and the second entry - for the right endpoint. Pass other boundary conditions as arrays of one element.

For the conditions defined as a combination of valid values, the library applies the following rules to identify the boundary condition type:

- If not required for spline construction, the value of boundary conditions is ignored.
- Not-a-knot condition has the highest priority. If set, other boundary conditions are ignored.
- Free-end condition has the second priority after the not-a-knot condition. If set, other boundary conditions are ignored.
- Periodic boundary condition has the next priority after the free-end condition.
- The first derivative has higher priority than the second derivative at the right and left endpoints.

If you set the periodic boundary condition, make sure that function values at the endpoints of the interpolation interval are identical. Otherwise, the library returns an error code. The table below specifies the values to be provided for each type of spline if the periodic boundary condition is set.

Boundary Requirements for Periodic Conditions

Spline Type	Periodic Boundary Condition Support	Boundary Value
Linear	Yes	Not required
Default quadratic	No	
Subbotin quadratic	No	
Natural cubic	Yes	Not required
Bessel	Yes	Not required
Akima	Yes	Not required
Hermite cubic	Yes	First derivative
Default cubic	Yes	Second derivative

Internal conditions supported in the Data Fitting domain that you can use for the `ic_type` parameter are the following:

Internal Conditions Supported by Data Fitting Functions

Internal Condition	Description	Spline
DF_NO_IC	No internal conditions provided.	
DF_IC_1ST_DER	Array of first derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Hermite cubic
DF_IC_2ND_DER	Array of second derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Default cubic
DF_IC_Q_KNOT	Knot array of size $n+1$, where n is the number of breakpoints.	Subbotin quadratic

To construct a Subbotin quadratic spline, you have three options to get the array of knots in the library:

- If you do not provide the knots, the library uses the default values of knots $t = \{t_i\}$, $i = 0, \dots, n$ according to the rule:

$$t_0 = x_0, t_n = x_{n-1}, t_i = (x_i + x_{i-1})/2, i = 1, \dots, n - 1.$$

- If you provide the knots in an array of size $n + 1$, the knots form a non-uniform partition. Make sure that the knot values you provide meet the following conditions:

$$t_0 = x_0, t_n = x_{n-1}, t_i \in (x_{i-1}, x_i), i = 1, \dots, n - 1.$$

- If you provide the knots in an array of size 4, the knots form a uniform partition

$$t_0 = x_0, t_1 = l, t_2 = r, t_3 = x_{n-1}, \text{ where } l \in (x_0, x_1) \text{ and } r \in (x_{n-2}, x_{n-1}).$$

In this case, you need to set the value of the `ic_type` parameter holding the type of internal conditions to `DF_IC_Q_KNOT` OR `DF_UNIFORM_PARTITION`.

NOTE

Since the partition is uniform, perform an OR operation with the `DF_UNIFORM_PARTITION` partition hint value described in [Table Hint Values for Partition x](#).

For computations based on look-up and step-wise constant interpolants, you can avoid calling the `df?editppspline1d` editor and directly call one of the routines for spline-based computation of spline values, derivatives, or integrals. For example, you can call the `df?construct1d` routine to construct the required spline with the given attributes, such as order or type.

The memory location of the spline coefficients is defined by the `scoeff` parameter. Make sure that the size of the array is sufficient to hold $ny*s_order * (nx-1)$ values.

The `df?editppspline1d` routine supports the following hint values for spline coefficients:

Hint Values for Spline Coefficients

Order	Description
DF_1ST_COORDINATE	The first coordinate of vector-valued data is provided.

Order	Description
DF_NO_HINT	No hint is provided. By default, all sets of spline coefficients are stored in row-major format.

The coefficients for all coordinates of the vector-valued function are packed in memory one by one in successive order, from function y_1 to function y_{ny} .

Within each coordinate, the library stores the coefficients as an array, in row-major format:

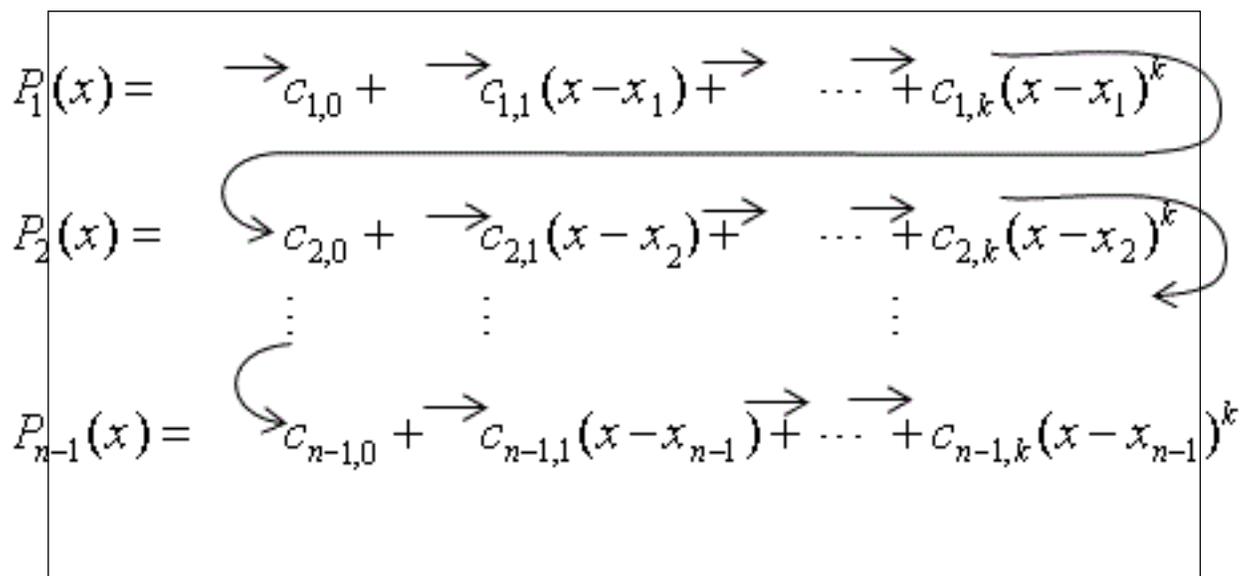
$$c_{1,0}, c_{1,1}, \dots, c_{1,k}, c_{2,0}, c_{2,1}, \dots, c_{2,k}, \dots, c_{n-1,0}, c_{n-1,1}, \dots, c_{n-1,k}$$

Mapping of the coefficients to storage in the `scoeff` array is described below, where $c_{i,j}$ is the j th coefficient of the function

$$P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$$

See [Mathematical Conventions](#) for more details on nomenclature and interpolants.

Row-major Coefficient Storage Format



If you store splines corresponding to different coordinates of the vector-valued function at non-contiguous memory locations, do the following:

1. Set the `scoeffhint` flag to `DF_1ST_COORDINATE` and provide the spline for the first coordinate.
2. Pass the spline coefficients for the remaining coordinates into the Data Fitting task using the `df?editidxptr` task editor.

Using the `df?editppsplined` task editor, you can provide to the Data Fitting task an already constructed spline that you want to use in computations. To ensure correct interpretation of the memory content, you should set the following parameters:

- Spline order and type, if appropriate. If the spline is not supported by the library, set the `s_type` parameter to `DF_PP_DEFAULT`.
- Pointer to the array of spline coefficients in row-major format.
- The `scoeffhint` parameter describing the structure of the array:
 - Set the `scoeffhint` flag to the `DF_1ST_COORDINATE` value to pass spline coefficients stored at different memory locations. In this case, you can set the parameters that describe boundary and internal conditions to zero.

- Use the default value `DF_NO_HINT` for all other cases.

Before passing an already constructed spline into the library, you should call the `dfieditval` task editor to provide the dimension of the spline `DF_NY`. See table "[Parameters Supported by the `dfieditval` Task Editor](#)" for details.

After you provide the spline to the Data Fitting task, you can run computations that use this spline.

NOTE

You must preserve the arrays `bc` (boundary conditions), `ic` (internal conditions), and `scoeff` (spline coefficients) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

`df?editptr`

Modifies a pointer to an array held in a Data Fitting task descriptor.

Syntax

```
status = dfseditptr(task, ptr_attr, ptr)
```

```
status = dfdeditptr(task, ptr_attr, ptr)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	TYPE (DF_TASK)	Descriptor of the task.
<code>ptr_attr</code>	INTEGER	The parameter to change. For details, see the <i>Pointer Attribute</i> column in table " Pointers Supported by the <code>df?editptr</code> Task Editor ".
<code>ptr</code>	REAL (KIND=4) DIMENSION (*) for <code>dfseditptr</code> REAL (KIND=8) DIMENSION (*) for <code>dfdeditptr</code>	New pointer. For details, see the <i>Purpose</i> column in table " Pointers Supported by the <code>df?editptr</code> Task Editor ".

Output Parameters

Name	Type	Description
<code>status</code>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?editptr` editor replaces the pointer of type `ptr_attr` stored in a Data Fitting task descriptor with a new pointer `ptr`. The table below describes types of pointers supported by the editor:

Pointers Supported by the `df?editptr` Task Editor

Pointer Attribute	Purpose
<code>DF_X</code>	Partition x of the interpolation interval, an array of strictly sorted breakpoints
<code>DF_Y</code>	Vector-valued function y
<code>DF_IC</code>	Internal conditions for spline construction. For details, see table " Internal Conditions Supported by Data Fitting Functions ".
<code>DF_BC</code>	Boundary conditions for spline construction. For details, see table " Boundary Conditions Supported by Data Fitting Functions ".
<code>DF_PP_SCOEFF</code>	Spline coefficients

You can use `df?editptr` to modify different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory. Use the `df?editidxptr` editor if you need to modify pointers to coordinates of the vector-valued function or spline coefficients stored at non-contiguous memory locations.

If you modify a partition of the interpolation interval, then you should call the `dfieditval` task editor with the corresponding value of `DF_XHINT`, even if the structure of the partition remains the same.

If you pass a `NULL` pointer to the `df?editptr` task editor, the task remains unchanged and the routine returns an error code. For the predefined error codes, please see "[Task Status and Error Reporting](#)".

NOTE

You must preserve the arrays x (breakpoints), y (vector-valued functions), bc (boundary conditions), ic (internal conditions), and $scoeff$ (spline coefficients) through the entire workflow of the Data Fitting computations which use those arrays, as the task stores the addresses of the arrays for spline-based computations.

`dfieditval`

Modifies a parameter value in a Data Fitting task descriptor.

Syntax

```
status = dfieditval(task, val_attr, val)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<code>TYPE(DF_TASK)</code>	Descriptor of the task.
<code>val_attr</code>	<code>INTEGER</code>	The parameter to change. See table " Parameters Supported by the <code>dfieditval</code> Task Editor ".
<code>val</code>	<code>INTEGER</code>	A new parameter value. See table " Parameters Supported by the <code>dfieditval</code> Task Editor ".

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `dfieditval` task editor replaces the parameter of type `val_attr` stored in a Data Fitting task descriptor with a new value `val`. The table below describes valid types of parameter `val_attr` supported by the editor:

Parameters Supported by the `dfieditval` Task Editor

Parameter Attribute	Purpose
<code>DF_NX</code>	Number of breakpoints
<code>DF_XHINT</code>	A flag describing the structure of partition. See table "Hint Values for Partition <i>x</i>" for the list of available values.
<code>DF_NY</code>	Dimension of the vector-valued function
<code>DF_YHINT</code>	A flag describing the structure of the vector-valued function. See table "Hint Values for Vector Function <i>y</i>" for the list of available values.
<code>DF_SPLINE_ORDER</code>	Spline order. See table "Spline Orders Supported by Data Fitting Functions" for the list of available values.
<code>DF_SPLINE_TYPE</code>	Spline type. See table "Spline Types Supported by Data Fitting Functions" for the list of available values.
<code>DF_BC_TYPE</code>	Type of boundary conditions used in spline construction. See table "Boundary Conditions Supported by Data Fitting Functions" for the list of available values.
<code>DF_IC_TYPE</code>	Type of internal conditions used in spline construction. See table "Internal Conditions Supported by Data Fitting Functions" for the list of available values.
<code>DF_PP_COEFF_HINT</code>	A flag describing the structure of spline coefficients. See table "Hint Values for Spline Coefficients" for the list of available values.
<code>DF_CHECK_FLAG</code>	A flag which controls checking of Data Fitting parameters. See table "Possible Values for the <code>DF_CHECK_FLAG</code> Parameter" for the list of available values.

If you pass a zero value for the parameter describing the size of the arrays that hold coefficients for a partition, a vector-valued function, or a spline, the parameter held in the Data fitting task remains unchanged and the routine returns an error code. For the predefined error codes, see ["Task Status and Error Reporting"](#).

Possible Values for the `DF_CHECK_FLAG` Parameter

Value	Description
<code>DF_ENABLE_CHECK_FLAG</code>	Checks the correctness of parameters of Data Fitting computational routines (default mode).
<code>DF_DISABLE_CHECK_FLAG</code>	Disables checking of the correctness of parameters of Data Fitting computational routines.

Use `DF_CHECK_FLAG` for `val_attr` in order to control validation of parameters of Data Fitting computational routines such as `df?construct1d`, `df?interpolate1d/df?interpolateex1d`, and `df?searchcells1d/df?searchcellsex1d`, which can perform better with a small number of interpolation sites or integration limits (fewer than one dozen). The default mode, with checking of parameters enabled, should be used as you develop a Data Fitting-based application. After you complete development you can disable parameter checking in order to improve the performance of your application.

If you modify the parameter describing dimensions of the arrays that hold the vector-valued function or spline coefficients in contiguous memory, you should call the `df?editptr` task editor with the corresponding pointers to the vector-valued function or spline coefficients even when this pointer remains unchanged. Call the `df?editidxptr` editor if those arrays are stored in non-contiguous memory locations.

You must call the `df?editval` task editor to edit the structure of the partition `DF_XHINT` every time you modify a partition using `df?editptr`, even if the structure of the partition remains the same.

`df?editidxptr`

Modifies a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

```
status = dfseditidxptr(task, ptr_attr, idx, ptr)
```

```
status = dfdeditidxptr(task, ptr_attr, idx, ptr)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<code>TYPE (DF_TASK)</code>	Descriptor of the task.
<code>ptr_attr</code>	<code>INTEGER</code>	Type of the data to be modified. The parameter takes one of the values described in "Data Attributes Supported by the <code>df?editidxptr</code> Task Editor".
<code>idx</code>	<code>INTEGER</code>	Index of the coordinate whose pointer is to be modified.
<code>ptr</code>	<code>REAL (KIND=4) DIMENSION (*)</code> for <code>dfseditidxptr</code> <code>REAL (KIND=8) DIMENSION (*)</code> for <code>dfdeditidxptr</code>	Pointer to the data that holds values of coordinate <code>idx</code> . For details, see table "Data Attributes Supported by the <code>df?editidxptr</code> Task Editor".

Output Parameters

Name	Type	Description
<code>status</code>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine modifies a pointer to the array that holds the `idx` coordinate of vector-valued function `y` or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the editor if you need to pass into a Data Fitting task or modify the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations. Do not use the editor for coordinates at contiguous memory locations in row-major format.

Before calling this editor, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific `df?editppspline1d` editor.

Data Attributes Supported by the `df?editidxptr` Task Editor

Data Attribute	Description
<code>DF_Y</code>	Vector-valued function <code>y</code>
<code>DF_PP_SCOEFF</code>	Piecewise polynomial spline coefficients

When using `df?editidxptr`, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension `ny` of the vector-valued function.
- You pass a `NULL` pointer to the editor. In this case, the task remains unchanged.
- You pass a pointer to the `idx` coordinate of the vector-valued function you provided to contiguous memory in column-major format.

The code example below demonstrates how to use the editor for providing values of a vector-valued function stored in two non-contiguous arrays:

`df?queryptr`

Reads a pointer to an array held in a Data Fitting task descriptor.

Syntax

```
status = dfsqueryptr(task, ptr_attr, ptr)
```

```
status = dfdqueryptr(task, ptr_attr, ptr)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	TYPE (DF_TASK)	Descriptor of the task.
<code>ptr_attr</code>	INTEGER	The parameter to query. The query routine supports pointer attributes described in the table " Pointers Supported by the df?editptr Task Editor ". For details, see the <i>Pointer Attribute</i> column in the table.

Output Parameters

Name	Type	Description
<code>ptr</code>	INTEGER (KIND=8)	Pointer to array returned by the query routine. For details, see the <i>Purpose</i> column in table " Pointers Supported by the df?editptr Task Editor ".
<code>status</code>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?queryptr` routine returns the pointer of type `ptr_attr` stored in a Data Fitting task descriptor as parameter `ptr`. Attributes of the pointers supported by the query function are identical to those supported by the editor `df?editptr` editor in the table "[Pointers Supported by the df?editptr Task Editor](#)".

You can use `df?queryptr` to read different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory.

dfiqueryval

Reads a parameter value in a Data Fitting task descriptor.

Syntax

```
status = dfiqueryval(task, val_attr, val)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	TYPE (DF_TASK)	Descriptor of the task.
<code>val_attr</code>	INTEGER	The parameter to query. The query function supports the parameter attributes described in " Parameters Supported by the dfieditval Task Editor ".

Output Parameters

Name	Type	Description
<code>val</code>	INTEGER	The parameter value returned by the query function. See table "Parameters Supported by the <code>dfieditval</code> Task Editor".
<code>status</code>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `dfiqueryval` routine returns a parameter of type `val_attr` stored in a Data Fitting task descriptor as parameter `val`. The query function supports the parameter attributes described in "Parameters Supported by the `dfieditval` Task Editor".

`df?queryidxptr`

Reads a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

```
status = dfsqueryidxptr(task, ptr_attr, idx, ptr)
```

```
status = dfdqueryidxptr(task, ptr_attr, idx, ptr)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	TYPE (DF_TASK)	Descriptor of the task.
<code>ptr_attr</code>	INTEGER	Pointer attribute to query. The parameter takes one of the attributes described in "Data Attributes Supported by the <code>df?editidxptr</code> Task Editor".
<code>idx</code>	INTEGER	Index of the coordinate of the pointer to query.

Output Parameters

Name	Type	Description
<code>ptr</code>	INTEGER (KIND=8)	Pointer to the data that holds values of coordinate <code>idx</code> returned. For details, see table "Data Attributes Supported by the <code>df?editidxptr</code> Task Editor".
<code>status</code>	INTEGER	Status of the routine:

Name	Type	Description
		<ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine returns a pointer to the array that holds the idx coordinate of vector-valued function y or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the query routine if you need the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations or at a contiguous memory location in row-major format (the default storage format for spline coefficients).

Before calling this query routine, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific `df?editppspline1d` editor.

When using `df?queryidxptr`, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension ny of the vector-valued function.
- You request the pointer to the idx coordinate of the vector-valued function you provided to contiguous memory in column-major format.

Computational Routines

Data Fitting computational routines are functions used to perform spline-based computations, such as:

- spline construction
- computation of values, derivatives, and integrals of the predefined order
- cell search

Once you create a Data Fitting task and initialize the required parameters, you can call computational routines as many times as necessary.

The table below lists the available computational routines:

Data Fitting Computational Routines

Routine	Description
<code>df?construct1d</code>	Constructs a spline for a one-dimensional Data Fitting task.
<code>df?interpolate1d</code>	Computes spline values and derivatives.
<code>df?interpolateex1d</code>	Computes spline values and derivatives by calling user-provided interpolants.
<code>df?integrate1d</code>	Computes spline-based integrals.
<code>df?integrateex1d</code>	Computes spline-based integrals by calling user-provided integrators.
<code>df?searchcells1d</code>	Finds indices of cells containing interpolation sites.
<code>df?searchcellsex1d</code>	Finds indices of cells containing interpolation sites by calling user-provided cell searchers.

If a Data Fitting computation completes successfully, the computational routines return the `DF_STATUS_OK` code. If an error occurs, the routines return an error code specifying the origin of the failure. Some possible errors are the following:

- The task pointer is `NULL`.
- Memory allocation failed.
- The computation failed for another reason.

For the list of available status codes, see ["Task Status and Error Reporting"](#).

NOTE

Data Fitting computational routines do not control errors for floating-point conditions, such as overflow, gradual underflow, or operations with Not a Number (NaN) values.

`df?construct1d`

Syntax

Constructs a spline of the given type.

```
status = dfsconstruct1d(task, s_format, method)
```

```
status = dfdconstruct1d(task, s_format, method)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<code>TYPE (DF_TASK)</code>	Descriptor of the task.
<code>s_format</code>	<code>INTEGER</code>	Spline format. The supported value is <code>DF_PP_SPLINE</code> .
<code>method</code>	<code>INTEGER</code>	Construction method. The supported value is <code>DF_METHOD_STD</code> .

Output Parameters

Name	Type	Description
<code>status</code>	<code>INTEGER</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

Before calling `df?construct1d`, you need to create and initialize the task, and set the parameters representing the spline. Then you can call the `df?construct1d` routine to construct the spline. The format of the spline is defined by parameter `s_format`. The method for spline construction is defined by parameter `method`. Upon successful construction, the spline coefficients are available in the user-provided memory location in the format you set through the Data Fitting editor. For the available storage formats, see table ["Hint Values for Spline Coefficients"](#).

df?interpolate1d/df?interpolateex1d

Runs data fitting computations.

Syntax

```
status = dfsinterpolate1d(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell)
```

```
status = dfdinterpolate1d(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell)
```

```
status = dfsinterpolateex1d(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params)
```

```
status = dfdinterpolateex1d(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params)
```

Include Files

- mkl_df.f90

Input Parameters

Name	Type	Description
<i>task</i>	TYPE (DF_TASK)	Descriptor of the task.
<i>type</i>	INTEGER	Type of spline-based computations. The parameter takes one or more values combined with an OR operation. For the list of possible values, see table " Computation Types Supported by the df?interpolate1d/ df?interpolate1d Routines ".
<i>method</i>	INTEGER	Computation method. The supported value is DF_METHOD_PP.
<i>nsite</i>	INTEGER	Number of interpolation sites.
<i>site</i>	REAL (KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL (KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	Array of interpolation sites of size <i>nsite</i> . The structure of the array is defined by the <i>sitehint</i> parameter: <ul style="list-style-type: none"> • If sites form a non-uniform partition, the array should contain <i>nsite</i> values. • If sites form a uniform partition, the array should contain two entries that represent the left and the right interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
<i>sitehint</i>	INTEGER	A flag describing the structure of the interpolation sites. For the list of possible values of <i>sitehint</i> , see table " Hint Values for Interpolation Sites ". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.

Name	Type	Description
<i>ndorder</i>	INTEGER	Maximal derivative order increased by one to be computed at interpolation sites.
<i>dorder</i>	INTEGER DIMENSION(*)	Array of size <i>ndorder</i> that defines the order of the derivatives to be computed at the interpolation sites. If all the elements in <i>dorder</i> are zero, the library computes the spline values only. If you do not need interpolation computations, set <i>ndorder</i> to zero and pass a NULL pointer to <i>dorder</i> .
<i>datahint</i>	REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	Array that contains additional information about the structure of partition <i>x</i> and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table " Structure of the <i>datahint</i> Array ".
<i>r</i>	REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d	Array for results. If you do not need spline-based interpolation or integration, set this pointer to NULL.
<i>rhint</i>	INTEGER	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table " Hint Values for the <i>rhint</i> Parameter ". If you set the flag to <code>DF_NO_HINT</code> , the library stores the result in row-major format.
<i>cell</i>	INTEGER DIMENSION(*)	Array of cell indices in partition <i>x</i> that contain the interpolation sites. Provide this parameter as input if <i>type</i> is <code>DF_INTERP_USER_CELL</code> . If you do not need cell indices, set this parameter to NULL.
<i>le_cb</i>	INTEGER	User-defined callback function for extrapolation at the sites to the left of the interpolation interval.
<i>le_params</i>	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function.
<i>re_cb</i>	INTEGER	User-defined callback function for extrapolation at the sites to the right of the interpolation interval.
<i>re_params</i>	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function.
<i>i_cb</i>	INTEGER	User-defined callback function for interpolation within the interpolation interval.

Name	Type	Description
<i>i_params</i>	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function.
<i>search_cb</i>	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites.
<i>search_params</i>	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> DF_STATUS_OK if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>		Contains results of computations at the interpolation sites.
<i>cell</i>		Array of cell indices in partition <i>x</i> that contain the interpolation sites, which is computed if <i>type</i> is DF_CELL.

Description

The `df?interpolate1d/df?interpolateex1d` routine performs spline-based computations with user-defined settings. The routine supports two types of computations for interpolation sites provided in array *site*:

Computation Types Supported by the `df?interpolate1d/df?interpolateex1d` Routines

Type	Description
DF_INTERP	Compute derivatives of predefined order. The derivative of the zero order is the spline value.
DF_INTERP_USER_CELL	Compute derivatives of predefined order given user-provided cell indices. The derivative of the zero order is the spline value. For this type of the computations you should provide a valid <i>cell</i> array, which holds the indices of cells in the <i>site</i> array containing relevant interpolation sites.
DF_CELL	Compute indices of cells in partition <i>x</i> that contain the sites.

If the indices of cells which contain interpolation types are available before the call to `df?interpolate1d/df?interpolateex1d`, you can improve performance by using the `DF_INTERP_USER_CELL` computation type.

NOTE

If you pass any combination of `DF_INTERP`, `DF_INTERP_USER_CELL`, and `DF_CELL` computation types to the routine, the library uses the `DF_INTERP_USER_CELL` computation mode.

If you specify `DF_INTERP_USER_CELL` computation mode and a user-defined callback function for computing cell indices to `df?interpolateex1d`, the library uses the `DF_INTERP_USER_CELL` computation mode, and the call-back function is not called.

If the sites do not belong to interpolation interval $[a, b]$, the library uses:

- polynomial P_0 of the spline constructed on interval $[x_0, x_1]$ for computations at the sites to the left of a .
- polynomial P_{n-2} of the spline constructed on interval $[x_{n-2}, x_{n-1}]$ for computations at the sites to the right of b .

Interpolation sites support the following hints:

Hint Values for Interpolation Sites

Value	Description
<code>DF_NON_UNIFORM_PARTITION</code>	Partition is non-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition is uniform.
<code>DF_SORTED_DATA</code>	Interpolation sites are sorted in the ascending order and define a non-uniform partition.
<code>DF_NO_HINT</code>	No hint is provided. By default, the partition defined by interpolation sites is interpreted as non-uniform.

NOTE

If you pass a sorted array of interpolation sites to the Intel MKL, set the `sitehint` parameter to the `DF_SORTED_DATA` value. The library uses this information when choosing the search algorithm and ignores any other data hints about the structure of the interpolation sites.

Data Fitting computation routines can use the following hints to speed up the computation:

- `DF_UNIFORM_PARTITION` describes the structure of breakpoints and the interpolation sites.
- `DF_QUASI_UNIFORM_PARTITION` describes the structure of breakpoints.

Pass the above hints to the library when appropriate.

The x pointer defines the memory location for the sets of interpolation and integration results for all coordinates of function y . The sets are stored one by one, in the successive order of the function coordinates from y_1 to y_{ny} .

You can define the following settings for packing the results within each set:

- Computation type: interpolation, integration, or both.
- Computation parameters: derivative orders.
- Storage format for the results. You can specify the format using the `rhint` parameter values described in the table below:

Hint Values for the `rhint` Parameter

Value	Description
<code>DF_MATRIX_STORAGE_ROWS</code>	Data is stored in row-major format according to C conventions.

Value	Description
DF_MATRIX_STORAGE_COLS	Data is stored in column-major format according to Fortran conventions.
DF_NO_HINT	No hint is provided. By default, the results are stored in row-major format.

For spline-based interpolation, you should set the derivatives whose values are required for the computation. You can provide the derivatives by setting the *dorder* array of size *ndorder* as follows:

$$dorder(i) = \begin{cases} 1, & \text{if derivative of the order } i-1 \text{ is required} \\ 0, & \text{otherwise} \end{cases} \quad i = 1, \dots, ndorder$$

See below a common structure of the storage formats of the interpolation results within each set *r* for computing derivatives of order i_1, i_2, \dots, i_m at *nsite* interpolation sites. In this description, *j* is the coordinate of the vector-valued function:

- Row-major format

$r_j(i_1, 1)$	$r_j(i_2, 1)$...	$r_j(i_m, 1)$
$r_j(i_1, 2)$	$r_j(i_2, 2)$...	$r_j(i_m, 2)$
...
$r_j(i_1, nsite)$	$r_j(i_2, nsite)$...	$r_j(i_m, nsite)$

- Column-major format

$r_j(i_1, 1)$	$r_j(i_1, 2)$...	$r_j(i_1, nsite)$
$r_j(i_2, 1)$	$r_j(i_2, 2)$...	$r_j(i_2, nsite)$
...
$r_j(i_m, 1)$	$r_j(i_m, 2)$...	$r_j(i_m, nsite)$

To speed up Data Fitting computations, use the *datahint* parameter that provides additional information about the structure of the partition and interpolation sites. This data represents a floating-point or a double array with the following structure:

Structure of the *datahint* Array

Element Number	Description
1	Task dimension
2	Type of additional information
3	Reserved field
4	The total number <i>q</i> of elements containing additional information.
5	Element (1)
...	...
<i>q</i> +4	Element (<i>q</i>)

Data Fitting computation functions support the following types of additional information for *datahint*(2):

Types of Additional Information

Type	Element Number	Parameter
DF_NO_APRIORI_INFO	0	No parameters are provided. Information about the data structure is absent.
DF_APRIORI_MOST_LIKELY_CELL	1	Index of the cell that is likely to contain interpolation sites.

To compute indices of the cells that contain interpolation sites, provide the pointer to the array of size $nsite$ for the results. The library supports the following scheme of cell indexing for the given partition $\{x_i\}$, $i=1, \dots, nx$:

$$cell(j) = i, \text{ if } site(j) \in [x_i, x_{i+1}), i = 0, \dots, nx,$$

where

- $x_0 = -\infty$
- $x_{nx+1} = +\infty$
- $j = 1, \dots, nsite$

To perform interpolation computations with spline types unsupported in the Data Fitting component, use the extended version of the routine `df?interpolateex1d`. With this routine, you can provide user-defined callback functions for computations within, to the left of, or to the right of interpolation interval $[a, b]$. The callback functions compute indices of the cells that contain the specified interpolation sites or can serve as an approximation for computing the exact indices of such cells.

If you do not pass any function for computations at the sites outside the interval $[a, b]$, the routine uses the default settings.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpcallback](#)

[df?searchcellscallback](#)

[df?integrate1d/df?integrateex1d](#)

Computes a spline-based integral.

Syntax

```
status = dfsintegrate1d(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)
```

```
status = dfdintegrate1d(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)
```

```
status = dfsintegrateex1d(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)
```

```
status = dfdintegrateex1d(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<i>task</i>	TYPE (DF_TASK)	Descriptor of the task.
<i>method</i>	INTEGER	Integration method. The supported value is DF_METHOD_PP.
<i>nlim</i>	INTEGER	Number of pairs of integration limits.
<i>llim</i>	REAL (KIND=4) DIMENSION (*) for dfsintegrate1d/ dfsintegrateex1d REAL (KIND=8) DIMENSION (*) for dfdintegrate1d/ dfdintegrateex1d	Array of size <i>nlim</i> that defines the left-side integration limits.
<i>llimhint</i>	INTEGER	A flag describing the structure of the left-side integration limits <i>llim</i> . For the list of possible values of <i>llimhint</i> , see table " Hint Values for Integration Limits ". If you set the flag to the DF_NO_HINT value, the library assumes that the left-side integration limits define a non-uniform partition.
<i>rlim</i>	REAL (KIND=4) DIMENSION (*) for dfsintegrate1d/ dfsintegrateex1d REAL (KIND=8) DIMENSION (*) for dfdintegrate1d/ dfdintegrateex1d	Array of size <i>nlim</i> that defines the right-side integration limits.
<i>rlimhint</i>	INTEGER	A flag describing the structure of the right-side integration limits <i>rlim</i> . For the list of possible values of <i>rlimhint</i> , see table " Hint Values for Integration Limits ". If you set the flag to the DF_NO_HINT value, the library assumes that the right-side integration limits define a non-uniform partition.
<i>ldatahint</i>	REAL (KIND=4) DIMENSION (*) for dfsintegrate1d/ dfsintegrateex1d REAL (KIND=8) DIMENSION (*) for dfdintegrate1d/ dfdintegrateex1d	Array that contains additional information about the structure of partition x and left-side integration limits. For details on the <i>ldatahint</i> array, see table " Structure of the datahint Array " in the description of the <code>df?intepolateld</code> function.
<i>rdatahint</i>	REAL (KIND=4) DIMENSION (*) for dfsintegrate1d/ dfsintegrateex1d	Array that contains additional information about the structure of partition x and right-side integration limits. For details on the <i>rdatahint</i> array, see table " Structure of the datahint Array " in the description of the <code>df?intepolateld</code> function.

Name	Type	Description
	REAL (KIND=8) DIMENSION (*) for dfdintegrate1d/ dfdintegrateex1d	
<i>rhint</i>	INTEGER	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table " Hint Values for Integration Results ". If you set the flag to the <code>DF_NO_HINT</code> value, the library stores the results in row-major format.
<i>le_cb</i>	INTEGER	User-defined callback function for integration on interval $[llim(i), \min(rlim(i), a)]$ for $llim(i) < a$.
<i>le_params</i>	INTEGER DIMENSION (*)	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function.
<i>re_cb</i>	INTEGER	User-defined callback function for integration on interval $[\max(llim(i), b), rlim(i)]$ for $rlim(i) \geq b$.
<i>re_params</i>	INTEGER DIMENSION (*)	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function.
<i>i_cb</i>	INTEGER	User-defined callback function for integration on interval $[\max(a, llim(i),), \min(rlim(i), b)]$.
<i>i_params</i>	INTEGER DIMENSION (*)	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function.
<i>search_cb</i>	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites.
<i>search_params</i>	INTEGER DIMENSION (*)	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>	REAL (KIND=4) DIMENSION (*) for dfsintegrate1d/ dfsintegrateex1d REAL (KIND=8) DIMENSION (*) for dfdintegrate1d/ dfdintegrateex1d	Array of integration results. The size of the array should be sufficient to hold $nlim \cdot ny$ values, where ny is the dimension of the vector-valued function. The integration results are packed according to the settings in <i>rhint</i> .

Description

The `df?integrate1d/df?integrateex1d` routine computes spline-based integral on user-defined intervals

$$I(i, j) = \int_{rl_i}^{ll_i} f_j(x) dx$$

where $rl_i = rlim(i)$, $ll_i = llim(i)$, and $i = 1, \dots, ny$.

If $rlim(i) < llim(i)$, the routine returns

$$I(i, j) = -\int_{ll_i}^{rl_i} f_j(x) dx$$

The routine supports the following hint values for integration results:

Hint Values for Integration Results

Value	Description
DF_MATRIX_STORAGE_ROWS	Data is stored in row-major format according to C conventions.
DF_MATRIX_STORAGE_COLS	Data is stored in column-major format according to Fortran conventions.
DF_NO_HINT	No hint is provided. By default, the coordinates of vector-valued function y are provided and stored in row-major format.

A common structure of the storage formats for the integration results is as follows:

- Row-major format

$I(1, 1)$...	$I(1, nlim)$
...
$I(ny, 1)$...	$I(ny, nlim)$

- Column-major format

$I(1, 1)$...	$I(ny, 1)$
...
$I(1, nlim)$...	$I(ny, nlim)$

Using the `llimhint` and `rlimhint` parameters, you can provide the following hint values for integration limits:

Hint Values for Integration Limits

Value	Description
DF_SORTED_DATA	Integration limits are sorted in the ascending order and define a non-uniform partition.
DF_NON_UNIFORM_PARTITION	Partition defined by integration limits is non-uniform.
DF_UNIFORM_PARTITION	Partition defined by integration limits is uniform.

Value	Description
DF_NO_HINT	No hint is provided. By default, partition defined by integration limits is interpreted as non-uniform.

To compute integration with splines unsupported in the Data Fitting component, use the extended version of the routine `df?integrateex1d`. With this routine, you can provide user-defined callback functions that compute:

- integrals within, to the left of, or to the right of the interpolation interval $[a, b]$
- indices of cells that contain the provided integration limits or can serve as an approximation for computing the exact indices of such cells

If you do not pass callback functions, the routine uses the default settings.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpolate1d/df?interpolateex1d](#)

[df?integrcallback](#)

[df?searchcellscallback](#)

[df?searchcells1d/df?searchcellsex1d](#)

Searches sub-intervals containing interpolation sites.

Syntax

```
status = dfssearchcells1d(task, method, nsite, site, sitehint, datahint, cell)
```

```
status = dfdsearchcells1d(task, method, nsite, site, sitehint, datahint, cell)
```

```
status = dfssearchcellsex1d(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)
```

```
status = dfdsearchcellsex1d(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<i>task</i>	TYPE (DF_TASK)	Descriptor of the task.
<i>method</i>	INTEGER	Search method. The supported value is <code>DF_METHOD_STD</code> .
<i>nsite</i>	INTEGER	Number of interpolation sites.
<i>site</i>	REAL (KIND=4) DIMENSION (*) for <code>dfssearchcells1d/</code> <code>dfssearchcellsex1d</code> REAL (KIND=8) DIMENSION (*) for <code>dfdsearchcells1d/</code> <code>dfdsearchcellsex1d</code>	<p>Array of interpolation sites of size <i>nsite</i>. The structure of the array is defined by the <i>sitehint</i> parameter:</p> <ul style="list-style-type: none"> • If the sites form a non-uniform partition, the array should contain <i>nsite</i> values. • If the sites form a uniform partition, the array should contain two entries that represent the left-most and the right-most interpolation sites. The first entry of the array

Name	Type	Description
		contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
<i>sitehint</i>	INTEGER	A flag describing the structure of the interpolation sites. For the list of possible values of <i>sitehint</i> , see table " Hint Values for Interpolation Sites ". If you set the flag to <code>DF_NO_HINT</code> , the library interprets the site-defined partition as non-uniform.
<i>datahint</i>	REAL(KIND=4) DIMENSION(*) for <code>dfssearchcells1d/dfssearchcellsex1d</code> REAL(KIND=8) DIMENSION(*) for <code>dfdsearchcells1d/dfdsearchcellsex1d</code>	Array that contains additional information about the structure of the partition and interpolation sites. This data helps to speed up the computation. If you provide a <code>NULL</code> pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table " Structure of the datahint Array ".
<i>search_cb</i>	INTEGER	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to <code>NULL</code> if you are not supplying a callback function.
<i>search_params</i>	INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to <code>NULL</code> if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	Status of the routine: <ul style="list-style-type: none"> <code>DF_STATUS_OK</code> if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>cell</i>	INTEGER DIMENSION(*)	Array of cell indices in the partition that contain the interpolation sites.

Description

The `df?searchcells1d/df?searchcellsex1d` routines return array *cell* of indices of sub-intervals (cells) in the partition that contain interpolation sites available in array *site*. For details on the cell indexing scheme, see the description of the `df?interpolate1d/df?interpolateex1d` computation routines.

Use the *datahint* parameter to provide additional information about the structure of the partition and/or interpolation sites. The definition of the *datahint* parameter is available in the description of the `df?interpolate1d/df?interpolateex1d` computation routines.

For description of the user-defined callback for computation of cell indices, see `df?searchcellscallback`.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpolate1d/df?interpolateex1d](#)
[df?searchcellscallback](#)

[df?interpcallback](#)

A callback function for user-defined interpolation to be passed into [df?interpolateex1d](#).

Syntax

```
status = dfsinterpcallback(n, cell, site, r, params)
```

```
status = dfdinterpcallback(n, cell, site, r, params)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER (KIND=8)	Number of interpolation sites.
<i>cell</i>	INTEGER (KIND=8) DIMENSION (*)	Array of size <i>n</i> containing indices of the cells to which the interpolation sites in array <i>site</i> belong.
<i>site</i>	REAL (KIND=4) DIMENSION (*) for <code>dfsinterpcallback</code> REAL (KIND=8) DIMENSION (*) for <code>dfdinterpcallback</code>	Array of interpolation sites of size <i>n</i> .
<i>params</i>	INTEGER DIMENSION (*)	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	The status returned by the callback function: <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. See " Task Status and Error Reporting " for error code definitions.
<i>r</i>	REAL (KIND=4) DIMENSION (*) for <code>dfsinterpcallback</code> REAL (KIND=8) DIMENSION (*) for <code>dfdinterpcallback</code>	Array of the computed interpolation results packed in row-major format.

Description

When passed into the [df?interpolateex1d](#) routine, this function performs user-defined interpolation operation.

See Also

[df?interpolate1d/df?interpolateex1d](#)
[df?searchcellscallback](#)

[df?integrcallback](#)

A callback function that you can pass into [df?integrateex1d](#) to define integration computations.

Syntax

```
status = dfsintegrcallback(n, lcell, llim, rcell, rlim, r, params)
status = dfdintegrcallback(n, lcell, llim, rcell, rlim, r, params)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER (KIND=8)	Number of pairs of integration limits.
<i>lcell</i>	INTEGER (KIND=8) DIMENSION (*)	Array of size <i>n</i> with indices of the cells that contain the left-side integration limits in array <i>llim</i> .
<i>llim</i>	REAL (KIND=4) DIMENSION (*) for <code>dfsintegrcallback</code> REAL (KIND=8) DIMENSION (*) for <code>dfdintegrcallback</code>	Array of size <i>n</i> that holds the left-side integration limits.
<i>rcell</i>	INTEGER (KIND=8) DIMENSION (*)	Array of size <i>n</i> with indices of the cells that contain the right-side integration limits in array <i>rlim</i> .
<i>rlim</i>	REAL (KIND=4) DIMENSION (*) for <code>dfsintegrcallback</code> REAL (KIND=8) DIMENSION (*) for <code>dfdintegrcallback</code>	Array of size <i>n</i> that holds the right-side integration limits.
<i>params</i>	INTEGER DIMENSION (*)	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	The status returned by the callback function: <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. See " Task Status and Error Reporting " for error code definitions.

Name	Type	Description
x	REAL(KIND=4) DIMENSION(*) for <code>dfsintegrcallback</code>	Array of integration results. For packing the results in row-major format, follow the instructions described in df?interpolate1d/df?interpolateex1d .
	REAL(KIND=8) DIMENSION(*) for <code>dfdintegrcallback</code>	

Description

When passed into the `df?integrateex1d` routine, this function defines integration computations. If at least one of the integration limits is outside the interpolation interval $[a, b]$, the library decomposes the integration into sub-intervals that belong to the extrapolation range to the left of a , the extrapolation range to the right of b , and the interpolation interval $[a, b]$, as follows:

- If the left integration limit is to the left of the interpolation interval ($llim < a$), the `df?integrateex1d` routine passes $llim$ as the left integration limit and $\min(rlim, a)$ as the right integration limit to the user-defined callback function.
- If the right integration limit is to the right of the interpolation interval ($rlim > b$), the `df?integrateex1d` routine passes $\max(llim, b)$ as the left integration limit and $rlim$ as the right integration limit to the user-defined callback function.
- If the left and the right integration limits belong to the interpolation interval, the `df?integrateex1d` routine passes them to the user-defined callback function unchanged.

The value of the integral is the sum of integral values obtained on the sub-intervals.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[df?integrate1d/df?integrateex1d](#)

[df?integrcallback](#)

[df?searchcellscallback](#)

[df?searchcellscallback](#)

A callback function for user-defined search to be passed into `df?interpolateex1d` or `df?searchcellsex1d`.

Syntax

```
status = dfssearchcellscallback(n, site, cell, flag, params)
```

```
status = dfdsearchcellscallback(n, site, cell, flag, params)
```

Include Files

- `mk1_df.f90`

Input Parameters

Name	Type	Description
<i>n</i>	INTEGER (KIND=8)	Number of interpolation sites.
<i>site</i>	REAL (KIND=4) DIMENSION (*) for <code>dfssearchcellscalback</code> REAL (KIND=8) DIMENSION (*) for <code>dfdsearchcellscalback</code>	Array of interpolation sites of size <i>n</i> .
<i>flag</i>	INTEGER (KIND=4) DIMENSION (*)	Array of size <i>n</i> , with values set as follows: <ul style="list-style-type: none"> • If the cell with index <code>cell(i)</code> contains <code>site(i)</code>, set <code>flag[i]</code> to 1. • Otherwise, set <code>flag(i)</code> to zero. In this case, the library interprets the index as an approximation and computes the index of the cell containing <code>site(i)</code> by using the provided index as a starting point for the search.
<i>params</i>	INTEGER DIMENSION (*)	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	INTEGER	The status returned by the callback function: <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • The <code>DF_STATUS_EXACT_RESULT</code> status indicates that cell indices returned by the callback function are exact. In this case, you do not need to initialize entries of the <code>flag</code> array. • A positive value indicates a warning. <p>See "Task Status and Error Reporting" for error code definitions.</p>
<i>cell</i>	INTEGER (KIND=8) DIMENSION (*)	Array of size <i>n</i> that returns indices of the cells computed by the callback function.

Description

When passed into the `df?interpolateex1d` or `df?searchcellsex1d` routine, this function performs a user-defined search.

See Also

[df?interpolate1d/df?interpolateex1d](#)
[df?interpcallback](#)

Task Destructors

Task destructors are routines used to delete task descriptors and deallocate the corresponding memory resources. The Data Fitting task destructor `dfdeletetask` destroys a Data Fitting task and frees the memory.

`dfdeletetask`

Destroys a Data Fitting task object and frees the memory.

Syntax

```
status = dfdeletetask(task)
```

Include Files

- `mkl_df.f90`

Input Parameters

Name	Type	Description
<code>task</code>	<code>TYPE (DF_TASK)</code>	Descriptor of the task to destroy.

Output Parameters

Name	Type	Description
<code>status</code>	<code>INTEGER</code>	Status of the routine: <ul style="list-style-type: none">• <code>DF_STATUS_OK</code> if the task is deleted successfully.• Non-zero error code if the operation failed. See "Task Status and Error Reporting" for error code definitions.

Description

Given a pointer to a task descriptor, this routine deletes the Data Fitting task descriptor and frees the memory allocated for the structure. If the task is deleted successfully, the routine sets the task pointer to `NULL`. Otherwise, the routine returns an error code.

Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an m -by- n matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in [Sparse Linear Systems](#) and various storage schemes for sparse matrices are described in [Sparse Matrix Storage Formats](#).

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix A . Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently, if an application involves well-conditioned matrices iterative solvers can be very efficient.

Direct Solvers, on the other hand, factor the matrix A into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array A .

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row i and column j in matrix A is denoted by $a(i, j)$. For example, a 3 by 4 matrix A , is written as follows:

$$A = \begin{bmatrix} a(1, 1) & a(1, 2) & a(1, 3) & a(1, 4) \\ a(2, 1) & a(2, 2) & a(2, 3) & a(2, 4) \\ a(3, 1) & a(3, 2) & a(3, 3) & a(3, 4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix A with the property that $a(i,j) = a(j,i)$, is called a symmetric matrix. A complex matrix A with the property that $a(i,j) = \text{conj}(a(j,i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix A is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i,j)$ and $a(j,i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix A are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T A x$ is greater than zero for all non-zero vectors x . A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A , the result of the matrix product PA is identical to A except for interchanging the rows of A . For a square matrix, it can be shown that if PA is a permutation of the rows of A , then AP^T is the same permutation of the columns of A . Additionally, it can be shown that the inverse of P is P^T .

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i -th row of a matrix to the j -th row, then the i -th element of the permutation vector is j .

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U , such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$\begin{aligned} Ax &= b \\ \Rightarrow LUx &= b \\ \Rightarrow L(Ux) &= b \end{aligned}$$

This leads to the following two-step process for finding the solution to the original system of equations:

- 1.** Solve the systems of equations $Ly = b$.
- 2.** Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A , that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H . Consequently, a solver can increase its efficiency by only storing L , and one-half of A , and not computing U . Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L , D , and half of A and we need not compute U . However, the backward solve phases must be amended to solving $L^T x = D^{-1} y$ rather than $L^T x = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is a symmetric positive definite sparse matrix, and A and b are defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & 1 & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & * & \frac{5}{8} & * \\ 3 & * & * & * & 16 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A . The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$L = \begin{bmatrix} \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

Notice that even though the matrix A is relatively sparse, the lower triangular matrix L has no zeros below the diagonal. If we computed L and then used it for the forward and backward solve phase, we would do as much computation as if A had been dense.

The situation of L having non-zeros in places where A has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of A in such a way as to reduce the fill-in when computing L . By doing this, the solver would only need to compute the non-zero entries in L . Toward this end, consider permuting the rows and columns of A . As described in [Matrix Fundamentals](#) section, the permutations of the rows of A can be represented as a permutation matrix, P . The result of permuting the rows is the product of P and A . Suppose, in the above example, we swap the first and fifth row

of A , then swap the first and fifth columns of A , and call the resulting matrix B . Mathematically, we can express the process of permuting the rows and columns of A to get B as $B = PAP^T$. After permuting the rows and columns of A , we see that B is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since B is obtained from A by simply switching rows and columns, the numbers of non-zero entries in A and B are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix}$$

The fill-in associated with B is much smaller than the fill-in associated with A . Consequently, the storage and computation time needed to factor B is much smaller than to factor A . Based on this, we see that an efficient sparse solver needs to find permutation P of the matrix A , which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of B to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general LU decomposition. Specifically, let P be a permutation matrix, $B = PAP^T$ and suppose that B can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow PA(P^{-1}P)x = Pb$$

$$\Rightarrow PA(P^T P)x = Pb$$

$$\Rightarrow (PAP^T)(Px) = Pb$$

$$\Rightarrow B(Px) = Pb$$

$$\Rightarrow LU(Px) = Pb$$

It follows that if we obtain an LU factorization for B , we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.
2. Solve $Uz = y$.
3. Set $x = P^Tz$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$Ly = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix} * \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

This gives: $y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{3}}{12}$.

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since a Cholesky factorization is used, $L^Tz = y$.

$$\begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix}^T * \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{3}}{12} \end{bmatrix}$$

This gives $z^T = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}$.

The third and final step is to set $x = P^T z$. This gives $x^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}$.

Sparse Matrix Storage Formats

As discussed above, it is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

Storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column (column-major format) or across each row (row-major format) in order, and writing the non-zero elements to a linear array in the order they appear in the walk.

- [DSS Symmetric Matrix Storage](#)
- [DSS Nonsymmetric Matrix Storage](#)
- [DSS Structurally Symmetric Matrix Storage](#)
- [DSS Distributed Symmetric Matrix Storage](#)

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

These sections describe in detail the sparse matrix storage formats supported in the current version of the Intel MKL Sparse BLAS Level 2 and Level 3.

- [Sparse BLAS CSR Matrix Storage](#)
- [Sparse BLAS CSC Matrix Storage](#)
- [Sparse BLAS Coordinate Matrix Storage](#)
- [Sparse BLAS Diagonal Matrix Storage](#)
- [Sparse BLAS Skyline Matrix Storage](#)
- [Sparse BLAS BSR Matrix Storage](#)

DSS Symmetric Matrix Storage

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solvers use a row-major upper triangular storage format: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored.

The Intel MKL sparse matrix storage format for direct sparse solvers is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

<i>values</i>	A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the <i>values</i> array using the row-major upper triangular storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column that contains the <i>i</i> -th element in the <i>values</i> array.

rowIndex Element *j* of the integer array *rowIndex* gives the index of the element in the *values* array that is first non-zero element in a row *j*.

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in the matrix.

As the *rowIndex* array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *i*-th row is equal to the difference of *rowIndex*(*i*) and *rowIndex*(*i*+1).

To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of *rowIndex*. Its value is equal to the number of non-zero elements plus one. This makes the total length of the *rowIndex* array one larger than the number of rows in the matrix.

NOTE

The Intel MKL sparse storage scheme for the direct sparse solvers supports both one-based indexing and zero-based indexing.

Consider the symmetric matrix *A*:

$$A = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{pmatrix}$$

Only elements from the upper triangle are stored. The actual arrays for the matrix *A* are as follows:

Storage Arrays for a Symmetric Matrix

one-based indexing										
<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(1	2	4	2	3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	8	9	10)			
zero-based indexing										
<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(0	1	3	1	2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	7	8	9)			

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Nonsymmetric Matrix Storage

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix *B*:

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

The matrix B has 13 non-zero elements, and all of them are stored as follows:

Storage Arrays for a Non-Symmetric Matrix

one-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>rowIndex</i>	=	(1	4	6	9	12	14)							
zero-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>rowIndex</i>	=	(0	3	5	8	11	13)							

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Structurally Symmetric Matrix Storage

Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if $a_{j,i}$ is not zero if and only if $a_{i,j}$ is not zero. From the point of view of the solver software, a "non-zero" element of a matrix is any element stored in the *values* array, even if its value is equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, the above matrix B can be turned into a symmetrically structured matrix by adding two non-zero entries:

$$B = \begin{pmatrix} 1 & -1 & * & 3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{pmatrix}$$

The matrix B can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Storage Arrays for a Symmetrically Structured Matrix

one-based indexing																
<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
<i>columns</i>	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
<i>rowIndex</i>	=	(1	4	7	10	13	16)									
zero-based indexing																
<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
<i>columns</i>	=	(0	1	3	0	1	4	2	3	4	0	2	3	1	2	4)
<i>rowIndex</i>	=	(0	3	6	9	12	15)									

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Distributed Symmetric Matrix Storage

The distributed assembled matrix input format can be used by the Parallel Direct Sparse Solver for Clusters Interface.

In this format, the symmetric input matrix *A* is divided into sequential row subsets, or domains. Each domain belongs to an MPI process. Neighboring domains can overlap. For such intersection between two domains, the element values of the full matrix can be obtained by summing the respective elements of both domains.

As in the centralized format, the distributed format uses three arrays to describe the input data, but the *values*, *columns*, and *rowIndex* arrays on each processor only describe the domain belonging to that particular processor and not the entire matrix.

For example, consider a symmetric matrix *A*:

$$A = \begin{pmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 11 & 5 & 4 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{pmatrix}$$

This array could be distributed between two domains corresponding to two MPI processes, with the first containing rows 1 through 3, and the second containing rows 3 through 5.

NOTE

For the symmetric input matrix, it is not necessary to store the values from the lower triangle.

$$A_{Domain1} = \begin{pmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 3 & * & 2 \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 1

one-based indexing							
<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(1	2	4	2	3	5)
<i>rowIndex</i>	=	(1	4	5	7)		
zero-based indexing							
<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(0	1	3	1	2	4)
<i>rowIndex</i>	=	(0	3	4	6)		

$$A_{Domain2} = \begin{pmatrix} * & * & 8 & 5 & 2 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 2

one-based indexing						
<i>values</i>	=	(8	5	2	10	5)
<i>columns</i>	=	(3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	6)	
zero-based indexing						
<i>values</i>	=	(8	5	2	10	5)

<i>columns</i>	=	(2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	5)	

The third row of matrix *A* is common between domain 1 and domain 2. The values of row 3 of matrix *A* are the sums of the respective elements of row 3 of matrix A_{Domain1} and row 1 of matrix A_{Domain2} .

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Sparse BLAS CSR Matrix Storage Format

The Intel MKL compressed sparse row (CSR) format is specified by four arrays: the *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the row-major storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> of <i>A</i> . Note that this index is equal to $pointerB(j) - pointerB(1) + 1$.
<i>pointerE</i>	An integer array that contains row indices, such that $pointerE(j) - pointerB(1)$ is the index of the element in the <i>values</i> array that is last non-zero element in a row <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.

NOTE

Note that the Intel MKL Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

The matrix *B*

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

can be represented in the CSR format as:

Storage Arrays for a Matrix in CSR Format

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>pointerB</i>	=	(1	4	6	9	12)								
<i>pointerE</i>	=	(4	6	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>pointerB</i>	=	(0	3	5	8	11)								
<i>pointerE</i>	=	(3	5	8	11	13)								

This storage format is used in the NIST Sparse BLAS library [Rem05].

Three Array Variation of CSR Format

The storage format accepted for the direct sparse solvers is a variation of the CSR format. It also is used in the Intel MKL Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix B can be represented in this format (referred to as the 3-array variation of the CSR format or CSR3) as:

Storage Arrays for a Matrix in CSR Format (3-Array Variation)

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)	
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)	
<i>rowIndex</i>	=	(1	4	6	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)	
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)	
<i>rowIndex</i>	=	(0	3	5	8	11	13)								

The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row J goes just after the set of non-zero elements in the row $J-1$.

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these arrays are pointers to the same array *values*.

By definition, the array *rowIndex* from the Table "Storage Arrays for a Non-Symmetric Example Matrix" is related to the arrays *pointerB* and *pointerE* from the Table "Storage Arrays for an Example Matrix in CSR Format", and you can see that

```
pointerB(i) = rowIndex(i) for i=1, ..5;
pointerE(i) = rowIndex(i+1) for i=1, ..5.
```

This enables calling a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

```
Subroutine name_routine(... , values, columns, pointerB, pointerE, ...)
```

can be called with parameters *values*, *columns*, *rowIndex* as follows:

```
call name_routine(... , values, columns, rowIndex, rowIndex(2), ...).
```

Sparse BLAS CSC Matrix Storage Format

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the column-major storage mapping.
<i>rows</i>	Element <i>i</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a column <i>j</i> of <i>A</i> . Note that this index is equal to $pointerB(j) - pointerB(1) + 1$.
<i>pointerE</i>	An integer array that contains column indices, such that $pointerE(j) - pointerB(1)$ is the index of the element in the <i>values</i> array that is last non-zero element in a column <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of columns in *A*.

NOTE

Note that the Intel MKL Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

For example, consider matrix *B*:

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

It can be represented in the CSC format as:

Storage Arrays for a Matrix in CSC Format

one-based indexing														
<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(1	2	4	1	2	5	3	4	1	3	4	3	5)
<i>pointerB</i>	=	(1	4	7	9	12)								
<i>pointerE</i>	=	(4	7	9	12	14)								
zero-based indexing														
<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(0	1	3	0	1	4	2	3	0	2	3	2	4)
<i>pointerB</i>	=	(0	3	6	8	11)								
<i>pointerE</i>	=	(3	6	8	11	13)								

Sparse BLAS Coordinate Matrix Storage Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel MKL coordinate format is specified by three arrays: *values*, *rows*, and *column*, and a parameter *nnz* which is number of non-zero elements in *A*. All three arrays have dimension *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> in any order.
<i>rows</i>	Element <i>i</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.

NOTE

Note that the Intel MKL Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix *C*

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

can be represented in the coordinate format as follows:

Storage Arrays for an Example Matrix in case of the coordinate format

one-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>rows</i>	=	(1	1	1	2	2	3	3	3	4	4	4	5	5)
<i>columns</i>	=	(1	2	3	1	2	3	4	5	1	3	4	2	5)
zero-based indexing														
<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>rows</i>	=	(0	0	0	1	1	2	2	2	3	3	3	4	4)
<i>columns</i>	=	(0	1	2	0	1	2	3	4	0	2	3	1	4)

Sparse BLAS Diagonal Matrix Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel MKL diagonal storage format is specified by two arrays: *values* and *distance*, and two parameters: *ndiag*, which is the number of non-empty diagonals, and *lval*, which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays *values* and *distance*:

<i>values</i>	A real or complex two-dimensional array is dimensioned as <i>lval</i> by <i>ndiag</i> . Each column of it contains the non-zero elements of certain diagonal of <i>A</i> . The key point of the storage is that each element in <i>values</i> retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of <i>distance</i> (<i>i</i>) is the number of elements to be padded for diagonal <i>i</i> .
<i>distance</i>	An integer array with dimension <i>ndiag</i> . Element <i>i</i> of the array <i>distance</i> is the distance between <i>i</i> -diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix C can be represented in the diagonal storage format as follows:

$$\begin{aligned}
 \text{distance} &= (-3 \quad -1 \quad 0 \quad 1 \quad 2) \\
 \text{values} &= \begin{pmatrix} * & * & 1 & -1 & -3 \\ * & -2 & 5 & 0 & 0 \\ * & 0 & 4 & 6 & 4 \\ -4 & 2 & 7 & 0 & * \\ 8 & 0 & -5 & * & * \end{pmatrix}
 \end{aligned}$$

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel MKL triangular solver routines elements of the array *distance* must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Sparse BLAS Skyline Matrix Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel MKL can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays: *values* and *pointers*. The following table describes these arrays:

<i>values</i>	A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
<i>pointers</i>	An integer array with dimension $(m+1)$, where m is the number of rows for lower triangle (columns for the upper triangle). $\text{pointers}(i) - \text{pointers}(1)+1$ gives the index of element in <i>values</i> that is first non-zero element in row (column) i . The value of $\text{pointers}(m+1)$ is set to $\text{nnz} + \text{pointers}(1)$, where nnz is the number of elements in the array <i>values</i> .

For example, consider the matrix C :

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

The low triangle of the matrix C given above can be stored as follows:

```

values = ( 1 -2 5 4 -4 0 2 7 8 0 0 -5 )
pointers = ( 1 2 4 5 9 13 )

```

and the upper triangle of this matrix C can be stored as follows:

```

values = ( 1 -1 5 -3 0 4 6 7 4 0 -5 )
pointers = ( 1 2 4 7 9 12 )

```

This storage format is supported by the NIST Sparse BLAS library [[Rem05](#)].

Note that the Intel MKL Sparse BLAS routines operating with the skyline storage format do not support general matrices.

Sparse BLAS BSR Matrix Storage Format

The Intel MKL block compressed sparse row (BSR) format for sparse matrices is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them are equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.
<i>pointerE</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that contains the last non-zero block in a row <i>j</i> of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of block rows in the block matrix.

NOTE

Note that the Intel MKL Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix *D*

$$D = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 1 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix *D* can be represented as a 3x3 block matrix *E* with the following structure:

$$E = \begin{pmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, N = \begin{pmatrix} 1 & 4 \\ 5 & 1 \end{pmatrix}, P = \begin{pmatrix} 4 & 3 \\ 0 & 0 \end{pmatrix}, Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$$

The matrix *D* can be represented in the BSR format as follows:

one-based indexing

```
values = (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0)
columns = (1 2 2 2 3)
pointerB = (1 3 4)
pointerE = (3 4 6)
```

zero-based indexing

```
values = [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
columns = [0 1 1 1 2]
pointerB = [0 2 3]
pointerE = [2 3 5]
```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Three Array Variation of BSR Format

Intel MKL supports the variation of the BSR format that is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>rowIndex</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks.

As the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the *i*-th row is equal to the difference of *rowIndex(i)* and *rowIndex(i+1)*.

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* with value equal to the number of non-zero blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrix.

The above matrix *D* can be represented in this 3-array variation of the BSR format as follows:

one-based indexing

```
values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)
columns = (1 2 2 2 3)
rowIndex = (1 3 4 6)
```

zero-based indexing

```
values = (1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0)
columns = (0 1 1 1 2)
rowIndex = (0 2 3 5)
```

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix F can be represented as a 3x3 block matrix G with the following structure:

$$G = \begin{pmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, M' = \begin{pmatrix} 6 & 8 \\ 7 & 2 \end{pmatrix}, N = \begin{pmatrix} 1 & 4 \\ 5 & 2 \end{pmatrix}, \text{ and } Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$$

The symmetric matrix F can be represented in this 3-array variation of the BSR format (storing only the upper triangular part) as follows:

one-based indexing

```
values = (1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0)
columns = (1 2 2 3)
rowIndex = (1 3 4 5)
```

zero-based indexing

```
values = (1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0)
columns = (0 1 1 2)
rowIndex = (0 2 3 4)
```

Variable BSR Format

A variation of BSR3 is variable block compressed sparse row format. For a trust level t , $0 \leq t \leq 100$, rows similar up to t percent are placed in one supernode.

Routine and Function Arguments

The major arguments in the BLAS routines are vector and matrix, whereas VM functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length n and increment $incx$ is passed in a one-dimensional array x whose values are defined as

```
x(1), x(1+|incx|), ..., x(1+(n-1)* |incx|)
```

If $incx$ is positive, then the elements in array x are stored in increasing order. If $incx$ is negative, the elements in array x are stored in decreasing order with the first element defined as $x(1+(n-1)* |incx|)$. If $incx$ is zero, then all elements of the vector have the same value, $x(1)$. The size of the one-dimensional array that stores the vector must always be at least

```
idimx = 1 + (n-1)* |incx |
```

Example. One-dimensional Real Array

Let $x(1:7)$ be the one-dimensional real array

```
x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0).
```

If $incx = 2$ and $n = 3$, then the vector argument with elements in order from first to last is (1.0, 5.0, 9.0).

If $incx = -2$ and $n = 4$, then the vector elements in order from first to last is (13.0, 9.0, 5.0, 1.0).

If $incx = 0$ and $n = 4$, then the vector elements in order from first to last is (1.0, 1.0, 1.0, 1.0).

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified.

In Fortran, storing the m -by- n matrix is based on column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is m , and the increment between elements on the same diagonal is $m + 1$.

Example. Two-dimensional Real Matrix

Let a be a real 5 x 4 matrix declared as `REAL A (5,4)float a[5*4];`

To scale the third column of a by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
call sscal (5, 2.0, a(1,3), 1)
```

To scale the second row, use the statement:

```
call sscal (4, 2.0, a(2,1), 5)
```

To scale the main diagonal of a by 2.0, use the statement:

```
call sscal (5, 2.0, a(1,1), 6)
```

NOTE

The default vector argument is assumed to be 1.

Vector Arguments in VM

Vector arguments of VM mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length n is passed contiguously in an array a whose values are defined as

$a(1), a(2), \dots, a(n)$

.

To accommodate for arrays with other increments, or more complicated indexing, VM contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array a as

$a(m_0), a(m_1), \dots, a(m_{n-1})$

and need to be regrouped into an array y as

$y(k_0), y(k_1), \dots, y(k_{n-1}),$

.

VM pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

$k_j = \text{incy} * j, m_j = \text{inca} * j, j = 1, \dots, n$

.

Constraint: $\text{incy} > 0$ and $\text{inca} > 0$.

For example, setting $\text{incy} = 1$ specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

$k_j = \text{iy}(j), m_j = \text{ia}(j), j = 1, \dots, n,$

.

where ia and iy are arrays of length n that contain index vectors for the input and output arrays a and y , respectively.

Mask Vector Indexing

Indices k_j, m_j are such that:

$m_y(k_j) \neq 0, m_a(m_j) \neq 0, j = 1, \dots, n,$

.

where m_a and m_y are arrays that contain mask vectors for the input and output arrays a and y , respectively.

Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- **conventional full storage**(in a two-dimensional array)
- **packed storage** for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- **band storage** for band matrices (in a two-dimensional array)
- **rectangular full packed storage** for symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels.

Full storage is the simplest scheme. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$, where lda is the leading dimension of array a .

If a matrix is triangular (upper or lower, as specified by the argument $uplo$), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if $uplo = 'U'$, a_{ij} is stored as described for $i \leq j$, other elements of a need not be set.

if $uplo = 'L'$, a_{ij} is stored as described for $j \leq i$, other elements of a need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument $uplo$) is packed by columns in a one-dimensional array ap :

if $uplo = 'U'$, a_{ij} is stored in $ap(i + j(j - 1)/2)$ for $i \leq j$

if $uplo = 'L'$, a_{ij} is stored in $ap(i + (2*n - j)*(j - 1)/2)$ for $j \leq i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in p .

Band storage is as follows: an m -by- n band matrix with kl non-zero sub-diagonals and ku non-zero super-diagonals is stored compactly in a two-dimensional array ab with $kl+ku + 1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

a_{ij} is stored in $ab(ku+1+i-j,j)$ for $\max(1,j-ku) \leq i \leq \min(n,j+kl)$.

Use the band storage scheme only when kl and ku are much less than the matrix size n . Although the routines work correctly for all values of kl and ku , using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when

$$m = n = 6, kl = 2, ku = 1$$

Array elements marked * are not used by the routines:

Banded matrix A	Band storage of A																								
$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$	<table border="0"> <tr> <td>*</td> <td>a_{12}</td> <td>a_{23}</td> <td>a_{34}</td> <td>a_{45}</td> <td>a_{56}</td> </tr> <tr> <td>a_{11}</td> <td>a_{22}</td> <td>a_{33}</td> <td>a_{44}</td> <td>a_{55}</td> <td>a_{66}</td> </tr> <tr> <td>a_{21}</td> <td>a_{32}</td> <td>a_{43}</td> <td>a_{54}</td> <td>a_{65}</td> <td>*</td> </tr> <tr> <td>a_{31}</td> <td>a_{42}</td> <td>a_{53}</td> <td>a_{64}</td> <td>*</td> <td>*</td> </tr> </table>	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*	a_{31}	a_{42}	a_{53}	a_{64}	*	*
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}																				
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}																				
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*																				
a_{31}	a_{42}	a_{53}	a_{64}	*	*																				

When a general band matrix is supplied for *LU factorization*, space must be allowed to store *kl* additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with *kl + ku* super-diagonals. Thus,

a_{ij} is stored in $ab(kl+ku+1+i-j, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

The band storage scheme for LU factorization is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

Banded matrix A	Band storage of A																																				
$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$	<table style="width: 100%; text-align: center; border-collapse: collapse;"> <tr> <td>*</td><td>*</td><td>*</td><td>+</td><td>+</td><td>+</td> </tr> <tr> <td>*</td><td>*</td><td>+</td><td>+</td><td>+</td><td>+</td> </tr> <tr> <td>*</td><td>a_{12}</td><td>a_{23}</td><td>a_{34}</td><td>a_{45}</td><td>a_{56}</td> </tr> <tr> <td>a_{11}</td><td>a_{22}</td><td>a_{33}</td><td>a_{44}</td><td>a_{55}</td><td>a_{66}</td> </tr> <tr> <td>a_{21}</td><td>a_{32}</td><td>a_{43}</td><td>a_{54}</td><td>a_{65}</td><td>*</td> </tr> <tr> <td>a_{31}</td><td>a_{42}</td><td>a_{53}</td><td>a_{64}</td><td>*</td><td>*</td> </tr> </table>	*	*	*	+	+	+	*	*	+	+	+	+	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*	a_{31}	a_{42}	a_{53}	a_{64}	*	*
*	*	*	+	+	+																																
*	*	+	+	+	+																																
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}																																
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}																																
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*																																
a_{31}	a_{42}	a_{53}	a_{64}	*	*																																

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

```

...|...
...|...
...|...

.....:.....:.....
.....:.....:.....:..
.....:.....:.....:..

```

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl=0$ if upper triangular, or $ku=0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* = 'U', a_{ij} is stored in $ab(k+1+i-j, j)$ for $\max(1, j-k) \leq i \leq j$

if *uplo* = 'L', a_{ij} is stored in $ab(1+i-j, j)$ for $j \leq i \leq \min(n, j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

- 'N' or 'n' normal (no conjugation, no transposition)
- 'T' or 't' transpose
- 'C' or 'c' conjugate transpose.

Example. Two-Dimensional Complex Array

Suppose $A(1:5, 1:4)$ is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54) \end{bmatrix}$$

Let *transa* be the transposition parameter, *m* be the number of rows, *n* be the number of columns, and *lda* be the leading dimension. Then if

transa = 'N', *m* = 4, *n* = 2, and *lda* = 5, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If *transa* = 'T', *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} \dots & (1.1, 0.11) & \dots & \dots & \dots & \dots \\ \dots & (2.1, 0.21) & \dots & \dots & \dots & \dots \\ \dots & (3.1, 0.31) & \dots & \dots & \dots & \dots \\ \dots & (4.1, 0.41) & \dots & \dots & \dots & \dots \end{bmatrix}$$

If *transa* = 'C', *m* = 4, *n* = 2, and *lda* =5, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array *A* above is declared as `COMPLEX A (5,4)`.

Then if `transa = 'N'`, `m = 3`, `n = 4`, and `lda = 4`, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

Rectangular Full Packed storage allows you to store symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels. To store an *n*-by-*n* triangle (and suppose for simplicity that *n* is even), you partition the triangle into three parts: two *n*/2-by-*n*/2 triangles and an *n*/2-by-*n*/2 square, then pack this as an *n*-by-*n*/2 rectangle (or *n*/2-by-*n* rectangle), by transposing (or transpose-conjugating) one of the triangles and packing it next to the other triangle. Since the two triangles are stored in full storage, you can use existing efficient routines on them.

There are eight cases of RFP storage representation: when *n* is even or odd, the packed matrix is transposed or not, the triangular matrix is lower or upper. See below for all the eight storage schemes illustrated:

n is odd, *A* is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
a ₁₁ X X X X X X	a ₁₁ a ₅₅ a ₆₅ a ₇₅	
a ₂₁ a ₂₂ X X X X X	a ₂₁ a ₂₂ a ₆₆ a ₇₆	a ₁₁ a ₂₁ a ₃₁ a ₄₁ a ₅₁ a ₆₁ a ₇₁
a ₃₁ a ₃₂ a ₃₃ X X X X	a ₃₁ a ₃₂ a ₃₃ a ₇₇	a ₅₅ a ₂₂ a ₃₂ a ₄₂ a ₅₂ a ₆₂ a ₇₂
a ₄₁ a ₄₂ a ₄₃ a ₄₄ X X X	a ₄₁ a ₄₂ a ₄₃ a ₄₄	a ₆₅ a ₆₆ a ₃₃ a ₄₃ a ₅₃ a ₆₃ a ₇₃
a ₅₁ a ₅₂ a ₅₃ a ₅₄ a ₅₅ X X	a ₅₁ a ₅₂ a ₅₃ a ₅₄	a ₇₅ a ₇₆ a ₇₇ a ₄₄ a ₅₄ a ₆₄ a ₇₄
a ₆₁ a ₆₂ a ₆₃ a ₆₄ a ₆₅ a ₆₆ X	a ₆₁ a ₆₂ a ₆₃ a ₆₄	
a ₇₁ a ₇₂ a ₇₃ a ₇₄ a ₇₅ a ₇₆ a ₇₇	a ₇₁ a ₇₂ a ₇₃ a ₇₄	

n is even, *A* is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
a ₁₁ X X X X X	a ₄₄ a ₅₄ a ₆₄	
a ₂₁ a ₂₂ X X X X	a ₁₁ a ₅₅ a ₆₅	
a ₃₁ a ₃₂ a ₃₃ X X X	a ₂₁ a ₂₂ a ₆₆	a ₄₄ a ₁₁ a ₂₁ a ₃₁ a ₄₁ a ₅₁ a ₆₁
a ₄₁ a ₄₂ a ₄₃ a ₄₄ X X	a ₃₁ a ₃₂ a ₃₃	a ₅₄ a ₅₅ a ₂₂ a ₃₂ a ₄₂ a ₅₂ a ₆₂
a ₅₁ a ₅₂ a ₅₃ a ₅₄ a ₅₅ X	a ₄₁ a ₄₂ a ₄₃	a ₆₄ a ₆₅ a ₆₆ a ₃₃ a ₄₃ a ₅₃ a ₆₃
a ₆₁ a ₆₂ a ₆₃ a ₆₄ a ₆₅ a ₆₆	a ₅₁ a ₅₂ a ₅₃	
	a ₆₁ a ₆₂ a ₆₃	

n is odd, *A* is upper triangular

Full format							RFP (not transposed)				RFP (transposed)													
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{14}	a_{15}	a_{16}	a_{17}														
X		a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{24}	a_{25}	a_{26}	a_{27}	a_{14}	a_{24}	a_{34}	a_{44}	a_{11}	a_{12}	a_{13}						
X	X		a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{34}	a_{35}	a_{36}	a_{37}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}	a_{22}	a_{23}						
X	X	X		a_{44}	a_{45}	a_{46}	a_{47}	a_{44}	a_{45}	a_{46}	a_{47}	a_{16}	a_{26}	a_{36}	a_{46}	a_{56}	a_{66}	a_{33}						
X	X	X	X		a_{55}	a_{56}	a_{57}	a_{11}	a_{55}	a_{56}	a_{57}	a_{17}	a_{27}	a_{37}	a_{47}	a_{57}	a_{67}	a_{77}						
X	X	X	X	X		a_{66}	a_{67}	a_{12}	a_{22}	a_{66}	a_{67}													
X	X	X	X	X	X		a_{77}	a_{13}	a_{23}	a_{33}	a_{77}													

n is even, A is upper triangular

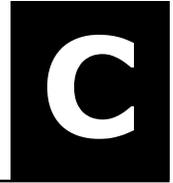
Full format						RFP (not transposed)			RFP (transposed)													
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{14}	a_{15}	a_{16}														
X		a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{24}	a_{25}	a_{26}	a_{14}	a_{24}	a_{34}	a_{44}	a_{11}	a_{12}	a_{13}						
X	X		a_{33}	a_{34}	a_{35}	a_{36}	a_{34}	a_{35}	a_{36}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}	a_{22}	a_{23}						
X	X	X		a_{44}	a_{45}	a_{46}	a_{44}	a_{45}	a_{46}	a_{16}	a_{26}	a_{36}	a_{46}	a_{56}	a_{66}	a_{33}						
X	X	X	X		a_{55}	a_{56}	a_{11}	a_{55}	a_{56}													
X	X	X	X	X		a_{66}	a_{12}	a_{22}	a_{66}													
							a_{13}	a_{23}	a_{33}													

Intel MKL provides a number of routines such as `?hfrk`, `?sfrk` performing BLAS operations working directly on RFP matrices, as well as some conversion routines, for instance, `?tpttf` goes from the standard packed format to RFP and `?trttf` goes from the full format to RFP.

Please refer to the Netlib site for more information.

Note that in the descriptions of LAPACK routines, arrays with RFP matrices have names ending in `fp`.

FFTW Interface to Intel® Math Kernel Library



Intel® Math Kernel Library (Intel® MKL) offers FFTW2 and FFTW3 interfaces to Intel MKL Fast Fourier Transform and Trigonometric Transform functionality. The purpose of these interfaces is to enable applications using FFTW (www.fftw.org) to gain performance with Intel MKL without changing the program source code.

Both FFTW2 and FFTW3 interfaces are provided in open source as FFTW wrappers to Intel MKL. For ease of use, FFTW3 interface is also integrated in Intel MKL.

Notational Conventions

This appendix typically employs path notations for Windows* OS.

FFTW2 Interface to Intel® Math Kernel Library

This section describes a collection of C and Fortran wrappers providing FFTW 2.x interface to Intel MKL. The wrappers translate calls to FFTW 2.x functions into the calls of the Intel MKL Fast Fourier Transform interface (FFT interface).

Note that Intel MKL FFT interface operates on both single- and double-precision floating-point data types.

Because of differences between FFTW and Intel MKL FFT functionalities, there are restrictions on using wrappers instead of the FFTW functions. Some FFTW functions have empty wrappers. However, many typical FFTs can be computed using these wrappers.

Refer to [chapter 11 "Fourier Transform Functions"](#), for better understanding the effects from the use of the wrappers.

Wrappers Reference

The section provides a brief reference for the FFTW 2.x C interface. For details please refer to the original FFTW 2.x documentation available at www.fftw.org.

Each FFTW function has its own wrapper. Some of them, which are *not* expressly listed in this section, are empty and do nothing, but they are provided to avoid link errors and satisfy the function calls.

See Also

[Limitations of the FFTW2 Interface to Intel MKL](#)

One-dimensional Complex-to-complex FFTs

The following functions compute a one-dimensional complex-to-complex Fast Fourier transform.

```
fftw_plan fftw_create_plan(int n, fftw_direction dir, int flags);
fftw_plan fftw_create_plan_specific(int n, fftw_direction dir, int flags, fftw_complex
*in, int istride, fftw_complex *out, int ostride);

void fftw(fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);

void fftw_one(fftw_plan plan, fftw_complex *in , fftw_complex *out);

void fftw_destroy_plan(fftw_plan plan);
```

Multi-dimensional Complex-to-complex FFTs

The following functions compute a multi-dimensional complex-to-complex Fast Fourier transform.

```
fftwnd_plan fftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
fftwnd_plan fftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);
fftwnd_plan fftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags,
fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags,
fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftwnd(fftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);

void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);

void fftwnd_destroy_plan(fftwnd_plan plan);
```

One-dimensional Real-to-half-complex/Half-complex-to-real FFTs

Half-complex representation of a conjugate-even symmetric vector of size N in a real array of the same size N consists of $N/2+1$ real parts of the elements of the vector followed by non-zero imaginary parts in the reverse order. Because the Intel MKL FFT interface does not currently support this representation, all wrappers of this kind are empty and do nothing.

Nevertheless, you can perform one-dimensional real-to-complex and complex-to-real transforms using `rfftwnd` functions with `rank=1`.

See Also

[Multi-dimensional Real-to-complex/Complex-to-real FFTs](#)

Multi-dimensional Real-to-complex/Complex-to-real FFTs

The following functions compute multi-dimensional real-to-complex and complex-to-real Fast Fourier transforms.

```
rfftwnd_plan rfftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
rfftwnd_plan rfftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);

rfftwnd_plan rfftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);

void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany, fftw_real *in, int istride,
int idist, fftw_complex *out, int ostride, int odist);

void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany, fftw_complex *in, int istride,
int idist, fftw_real *out, int ostride, int odist);

void rfftwnd_one_real_to_complex(rfftwnd_plan plan, fftw_real *in, fftw_complex *out);
```

```
void rffftwnd_one_complex_to_real(rffftwnd_plan plan, fftw_complex *in, fftw_real *out);
void rffftwnd_destroy_plan(rffftwnd_plan plan);
```

Multi-threaded FFTW

This section discusses multi-threaded FFTW wrappers only. MPI FFTW wrappers, available only with Intel MKL for the Linux* and Windows* operating systems, are described in [section "MPI FFTW Wrappers"](#).

Unlike the original FFTW interface, every computational function in the FFTW2 interface to Intel MKL provides multithreaded computation by default, with the maximum number of threads permitted in FFT functions (see "Techniques to Set the Number of Threads" in *Intel MKL User's Guide*). To limit the number of threads, call the threaded FFTW computational functions:

```
void fftw_threads(int nthreads, fftw_plan plan, int howmany, fftw_complex *in, int
istride, int idist, fftw_complex *out, int ostride, int odist);
void fftw_threads_one(int nthreads, rffftwnd_plan plan, fftw_complex *in, fftw_complex
*out);
...
void rffftwnd_threads_real_to_complex( int nthreads, rffftwnd_plan plan, int howmany,
fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int odist);
```

Compared to its non-threaded counterpart, every threaded computational function has `threads_` as the second part of its name and additional first parameter `nthreads`. Set the `nthreads` parameter to the thread limit to ensure that the computation requires at most that number of threads.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

FFTW Support Functions

The FFTW wrappers provide memory allocation functions to be used with FFTW:

```
void* fftw_malloc(size_t n);
void fftw_free(void* x);
```

The `fftw_malloc` wrapper aligns the memory on a 16-byte boundary.

If `fftw_malloc` fails to allocate memory, it aborts the application. To override this behavior, set a global variable `fftw_malloc_hook` and optionally the complementary variable `fftw_free_hook`:

```
void (*fftw_malloc_hook) (size_t n);
void (*fftw_free_hook) (void *p);
```

The wrappers use the function `fftw_die` to abort the application in cases when a caller cannot be informed of an error otherwise (for example, in computational functions that return `void`). To override this behavior, set a global variable `fftw_die_hook`:

```
void (*fftw_die_hook) (const char *error_string);
void fftw_die(const char *s);
```

Calling Wrappers from Fortran

The FFTW2 wrappers to Intel MKL provide the following subroutines for calling from Fortran:

```
call fftw_f77_create_plan(plan, n, dir, flags)
call fftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftw_f77_one(plan, in, out)
call fftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call fftw_f77_threads_one(nthreads, plan, in, out)
call fftw_f77_destroy_plan(plan)

call fftwnd_f77_create_plan(plan, rank, n, dir, flags)
call fftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call fftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call fftwnd_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftwnd_f77_one(plan, in, out)
call fftwnd_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call fftwnd_f77_threads_one(nthreads, plan, in, out)
call fftwnd_f77_destroy_plan(plan)

call rfftw_f77_create_plan(plan, n, dir, flags)
call rfftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call rfftw_f77_one(plan, in, out)
call rfftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftw_f77_threads_one(nthreads, plan, in, out)
call rfftw_f77_destroy_plan(plan)

call rfftwnd_f77_create_plan(plan, rank, n, dir, flags)
call rfftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call rfftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call rfftwnd_f77_complex_to_real(plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftwnd_f77_one_complex_to_real (plan, in, out)
call rfftwnd_f77_real_to_complex(plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftwnd_f77_one_real_to_complex (plan, in, out)
call rfftwnd_f77_threads_complex_to_real(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_complex_to_real(nthreads, plan, in, out)
call rfftwnd_f77_threads_real_to_complex(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_real_to_complex(nthreads, plan, in, out)
```

```
call rfftwnd_f77_destroy_plan(plan)
```

```
call fftw_f77_threads_init(info)
```

The FFTW Fortran functions are wrappers to FFTW C functions.

See also these resources:

www.fftw.org for the original FFTW 2.x documentation.

[Limitations of the FFTW2 Interface to Intel MKL](#) for limitations of the wrappers.

Limitations of the FFTW2 Interface to Intel MKL

The FFTW2 wrappers implement the functionality of only those FFTW functions that Intel MKL can reasonably support. Other functions are provided as no-operation functions, whose only purpose is to satisfy link-time symbol resolution. Specifically, no-operation functions include:

- Real-to-half-complex and respective backward transforms
- Print plan functions
- Functions for importing/exporting/forgetting wisdom
- Most of the FFTW functions not covered by the original FFTW2 documentation

Because the Intel MKL implementation of FFTW2 wrappers does not use plan and plan node structures declared in `fftw.h`, the behavior of an application that relies on the internals of the plan structures defined in that header file is undefined.

FFTW2 wrappers define plan as a set of attributes, such as strides, used to commit the Intel MKL FFT descriptor structure. If an FFTW2 computational function is called with attributes different from those recorded in the plan, the function attempts to adjust the attributes of the plan and recommit the descriptor. So, repeated calls of a computational function with the same plan but different strides, distances, and other parameters may be performance inefficient.

Plan creation functions disregard most planner flags passed through the `flags` parameter. These functions take into account only the following values of `flags`:

- `FFTW_IN_PLACE`
If this value of `flags` is supplied, the plan is marked so that computational functions using that plan ignore the parameters related to output (`out`, `ostride`, and `odist`). Unlike the original FFTW interface, the wrappers never use the `out` parameter as a scratch space for in-place transforms.
- `FFTW_THREADSAFE`
If this value of `flags` is supplied, the plan is marked read-only. An attempt to change attributes of a read-only plan aborts the application.

FFTW wrappers are generally not thread safe. Therefore, do not use the same plan in parallel user threads simultaneously.

Installation

Wrappers are delivered as source code, which you must compile to build the wrapper library. Then you can substitute the wrapper and Intel MKL libraries for the FFTW library. The source code for the wrappers, makefiles, and files with lists of wrappers are located in the `.\interfaces\fftw2xf` subdirectory in the Intel MKL directory.

Creating the Wrapper Library

Three header files are used to compile the Fortran wrapper library: `fftw2_mkl.h`, `fftw2_f77_mkl.h`, and `fftw.h`. The `fftw2_mkl.h` and `fftw2_f77_mkl.h` files are located in the `.\interfaces\fftw2xf\wrappers` subdirectory in the Intel MKL directory.

The file `fftw.h`, used to compile libraries and located in the `.\include\fftw` subdirectory in the Intel MKL directory, slightly differs from the original FFTW (www.fftw.org) header file `fftw.h`.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw2xf` subdirectory in the Intel MKL directory.

A wrapper library contains wrappers for complex and real transforms in a serial and multi-threaded mode for double- or single-precision floating-point data types. A makefile parameter manages the data type.

Parameters of a makefile also specify the platform (required), compiler, and data precision. The makefile comment heading provides the exact description of these parameters.

Because a C compiler builds the Fortran wrapper library, function names in the wrapper library and Fortran object module may be different. The file `fftw2_f77_mkl.h` in the `.\interfaces\fftw2xf\source` subdirectory in the Intel MKL directory defines function names according to the names in the Fortran module. If a required name is missing in the file, you can modify the file to add the name before building the library.

To build the library, run the `make` command on Linux* OS and OS X* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using the Intel® C++ Compiler or the Intel® Fortran Compiler (Compilers and data precision are chosen by default.)

Each makefile creates the library in the directory with Intel MKL libraries corresponding to the platform used. For example, `./lib/ia32` (on Linux OS and OS X) or `.\lib\ia32` (on Windows* OS).

In the names of a wrapper library, the suffix corresponds to the compiler used and the letter preceding the underscore is "f" for the Fortran programming language.

For example,

```
fftw2xf_intel.lib (on Windows OS); libfftw2xf_intel.a (on Linux OS and OS X);
```

Application Assembling

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created wrapper library and the Intel MKL library instead of the FFTW library.

Running Examples

Intel MKL provides examples to demonstrate how to use the MPI FFTW wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw2xf` subdirectory in the Intel MKL directory. To build examples, several additional files are needed: `fftw.h`, `fftw_threads.h`, `rfftw.h`, `rfftw_threads.h`, and `fftw_f77.i`. These files are distributed with permission from FFTW and are available in `.\include\fftw`. The original files can also be found in FFTW 2.1.5 at <http://www.fftw.org/download.html>.

An example makefile uses the `function` parameter in addition to the parameters of the corresponding wrapper library makefile (see [Creating a Wrapper Library](#)). The makefile comment heading provides the exact description of these parameters.

An example makefile normally invokes examples. However, if the appropriate wrapper library is not yet created, the makefile first builds the library the same way as the wrapper library makefile does and then proceeds to examples.

If the parameter `function=<example_name>` is defined, only the specified example runs. Otherwise, all examples from the appropriate subdirectory run. The subdirectory `._results` is created, and the results are stored there in the `<example_name>.res` files.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

FFTW3 Interface to Intel® Math Kernel Library

This section describes a collection of FFTW3 wrappers to Intel MKL. The wrappers translate calls of FFTW3 functions to the calls of the Intel MKL Fourier transform (FFT) or Trigonometric Transform (TT) functions. The purpose of FFTW3 wrappers is to enable developers whose programs currently use the FFTW3 library to gain performance with the Intel MKL Fourier transforms without changing the program source code.

The FFTW3 wrappers provide a limited functionality compared to the original FFTW 3.x library, because of differences between FFTW and Intel MKL FFT and TT functionality. This section describes limitations of the FFTW3 wrappers and hints for their usage. Nevertheless, many typical FFT tasks can be performed using the FFTW3 wrappers to Intel MKL.

The FFTW3 wrappers are integrated in Intel MKL. The only change required to use Intel MKL through the FFTW3 wrappers is to link your application using FFTW3 against Intel MKL.

A reference implementation of the FFTW3 wrappers is also provided in open source. You can find it in the `interfaces` directory of the Intel MKL distribution. You can use the reference implementation to create your own wrapper library (see [Building Your Own Wrapper Library](#))

See also these resources:

Intel MKL Release Notes for the version of the FFTW3 library supported by the wrappers.

www.fftw.org for a description of the FFTW interface.

[Fourier Transform Functions](#) for a description of the Intel MKL FFT interface.

[Trigonometric Transform Routines](#) for a description of Intel MKL TT interface.

Using FFTW3 Wrappers

The FFTW3 wrappers are a set of functions and data structures depending on one another. The wrappers are not designed to provide the interface on a function-per-function basis. Some FFTW3 wrapper functions are empty and do nothing, but they are present to avoid link errors and satisfy function calls.

This manual does not list the declarations of the functions that the FFTW3 wrappers provide (you can find the declarations in the `fftw3.h` header file). Instead, this section comments particular limitations of the wrappers and provides usage hints:

- The FFTW3 wrappers do not support long double precision because Intel MKL FFT functions operate only on single- and double-precision floating-point data types. Therefore the functions with prefix `fftwl_`, supporting the `long double` data type, are not provided.
- The wrappers provide equivalent implementation for double- and single-precision functions (those with prefixes `fftw_` and `fftwf_`, respectively). So, all these comments equally apply to the double- and single-precision functions and will refer to functions with prefix `fftw_`, that is, double-precision functions, for brevity.

- The FFTW3 interface that the wrappers provide is defined in the `fftw3.h` and `fftw3.f` header files. These files are borrowed from the FFTW3.x package and distributed within Intel MKL with permission. Additionally, the `fftw3_mkl.h`, `fftw3_mkl.f`, and `fftw3_mkl_f77.h` header files define supporting structures and supplementary constants and macros, as well as expose Fortran interface in C.
- Actual functionality of the plan creation wrappers is implemented in `guru64` set of functions. Basic interface, advanced interface, and `guru` interface plan creation functions call the `guru64` interface functions. So, all types of the FFTW3 plan creation interface in the wrappers are functional.
- Plan creation functions may return a `NULL` plan, indicating that the functionality is not supported. So, please carefully check the result returned by plan creation functions in your application. In particular, the following problems return a `NULL` plan:
 - `c2r` and `r2c` problems with a split storage of complex data.
 - `r2r` problems with `kind` values `FFTW_R2HC`, `FFTW_HC2R`, and `FFTW_DHT`. The only supported `r2r` kinds are even/odd DFTs (sine/cosine transforms).
 - Multidimensional `r2r` transforms.
 - Transforms of multidimensional vectors. That is, the only supported values for parameter `howmany_rank` in `guru` and `guru64` plan creation functions are 0 and 1.
 - Multidimensional transforms with `rank > MKL_MAXRANK`.
- The `MKL_RODFT00` value of the `kind` parameter is introduced by the FFTW3 wrappers. For better performance, you are strongly encouraged to use this value rather than `FFTW_RODFT00`. To use this `kind` value, provide an extra first element equal to 0.0 for the input/output vectors. Consider the following example:

```
plan1 = fftw_plan_r2r_1d(n, in1, out1, FFTW_RODFT00, FFTW_ESTIMATE);
plan2 = fftw_plan_r2r_1d(n, in2, out2, MKL_RODFT00, FFTW_ESTIMATE);
```

Both plans perform the same transform, except that the `in2/out2` arrays have one extra zero element at location 0. For example, if `n=3`, `in1={x,y,z}` and `out1={u,v,w}`, then `in2={0,x,y,z}` and `out2={0,u,v,w}`.

- The `flags` parameter in plan creation functions is always ignored. The same algorithm is used regardless of the value of this parameter. In particular, `flags` values `FFTW_ESTIMATE`, `FFTW_MEASURE`, etc. have no effect.
- For multithreaded plans, use normal sequence of calls to the `fftw_init_threads()` and `fftw_plan_with_nthreads()` functions (refer to FFTW documentation).
- Memory allocation function `fftw_malloc` returns memory aligned at a 16-byte boundary. You must free the memory with `fftw_free`.
- The FFTW3 wrappers to Intel MKL use the 32-bit `int` type in both LP64 and ILP64 interfaces of Intel MKL. Use `guru64` FFTW3 interfaces for 64-bit sizes.
- Fortran wrappers (see [Calling Wrappers from Fortran](#)) use the `INTEGER` type, which is 32-bit in LP64 interfaces and 64-bit in ILP64 interfaces.
- The wrappers typically indicate a problem by returning a `NULL` plan. In a few cases, the wrappers may report a descriptive message of the problem detected. By default the reporting is turned off. To turn it on, set variable `fftw3_mkl.verbose` to a non-zero value, for example:

```
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.verbose = 0;
plan = fftw_plan_r2r(...);
```

- The following functions are empty:
 - For saving, loading, and printing plans
 - For saving and loading wisdom
 - For estimating arithmetic cost of the transforms.
- Do not use macro `FFTW_DLL` with the FFTW3 wrappers to Intel MKL.
- Do not use negative stride values. Though FFTW3 wrappers support negative strides in the part of advanced and `guru` FFTW interface, the underlying implementation does not.

Calling Wrappers from Fortran

Intel MKL also provides Fortran 77 interfaces of the FFTW3 wrappers. The Fortran wrappers are available for all FFTW3 interface functions and are based on C interface of the FFTW3 wrappers. Therefore they have the same functionality and restrictions as the corresponding C interface wrappers.

The Fortran wrappers use the default `INTEGER` type for integer arguments. The default `INTEGER` is 32-bit in Intel MKL LP64 interfaces and 64-bit in ILP64 interfaces. Argument `plan` in a Fortran application must have type `INTEGER*8`.

The wrappers that are double-precision subroutines have prefix `dfftw_`, single-precision subroutines have prefix `sfftw_` and provide an equivalent functionality. Long double subroutines (with prefix `lfftw_`) are not provided.

The Fortran FFTW3 wrappers use the default Intel® Fortran compiler convention for name decoration. If your compiler uses a different convention, or if you are using compiler options affecting the name decoration (such as `/Qlowercase`), you may need to compile the wrappers from sources, as described in section [Building Your Own Wrapper Library](#).

For interoperability with C, the declaration of the Fortran FFTW3 interface is provided in header file `include/fftw/fftw3_mkl_f77.h`.

You can call Fortran wrappers from a FORTRAN 77 or Fortran 90 application, although Intel MKL does not provide a Fortran 90 module for the wrappers. For a detailed description of the FFTW Fortran interface, refer to FFTW3 documentation (www.fftw.org).

The following example illustrates calling the FFTW3 wrappers from Fortran:

```
INTEGER*8 plan
INTEGER N
INCLUDE 'fftw3.f'
COMPLEX*16 IN(*), OUT(*)
!...initialize array IN
CALL DFFTW_PLAN_DFT_1D(PLAN, N, IN, OUT, -1, FFTW_ESTIMATE)
IF (PLAN .EQ. 0) STOP
CALL DFFTW_EXECUTE
!...result is in array OUT
```

Building Your Own Wrapper Library

The FFTW3 wrappers to Intel MKL are delivered both integrated in Intel MKL and as source code, which can be compiled to build a standalone wrapper library with exactly the same functionality. Normally you do not need to build the wrappers yourself. However, if your Fortran application is compiled with a compiler that uses a different name decoration than the Intel® Fortran compiler or if you are using compiler options altering the Fortran name decoration, you may need to build the wrappers that use the appropriate name changing convention.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw3xf` subdirectory in the Intel MKL directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS and OS X* or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default integer type, and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the FFTW3 Fortran wrappers to Intel MKL for use from the GNU g77* Fortran compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3xf
make libintel64 compiler=gnu fname=a_name__ INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler, decorates the name with the second underscore, and places the result, named `libfftw3xf_gnu.a`, into the `/my/path` directory. The name of the resulting library is composed of the name of the compiler used and may be changed by an optional parameter `INSTALL_LIBNAME`.

Building an Application

Normally, the only change needed to build your application with FFTW3 wrappers replacing original FFTW library is to add Intel MKL at the link stage (see section "Linking Your Application with Intel® Math Kernel Library" in the Intel MKL User's Guide).

If you recompile your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Sometimes, you may have to modify your application according to the following recommendations:

- The application requires

```
#include "fftw3.h",
```

which it probably already includes.

- The application does not require

```
#include "mkl_dfti.h" .
```

- The application does not require

```
#include "fftw3_mkl.h" .
```

It is required only in case you want to use the `MKL_RODFT00` constant.

- If the application does not check whether a `NULL` plan is returned by plan creation functions, this check must be added, because the FFTW3 to Intel MKL wrappers do not provide 100% of FFTW3 functionality.

Running Examples

There are some examples that demonstrate how to use the wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xf` subdirectory in the Intel MKL directory. To build Fortran examples, one additional file `fftw3.f` is needed. This file is distributed with permission from FFTW and is available in the `.\include\fftw` subdirectory of the Intel MKL directory. The original file can also be found in FFTW 3.3.4 at <http://www.fftw.org/download.html>.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. If the parameter `function=<example_name>` is defined, then only the specified example will run. Otherwise, all examples will be executed. Results of running the examples are saved in subdirectory `._results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

MPI FFTW Wrappers

This section describes a collection of MPI FFTW wrappers to Intel® MKL.

MPI FFTW wrappers are available only with Intel MKL for the Linux* and Windows* operating systems.

These wrappers translate calls of MPI FFTW functions to the calls of the Intel MKL cluster Fourier transform (CFFT) functions. The purpose of the wrappers is to enable users of MPI FFTW functions improve performance of the applications without changing the program source code.

Although the MPI FFTW wrappers provide less functionality than the original FFTW3 because of differences between MPI FFTW and Intel MKL CFFT, the wrappers cover many typical CFFT use cases.

The MPI FFTW wrappers are provided as source code. To use the wrappers, you need to build your own wrapper library (see [Building Your Own Wrapper Library](#)).

See also these resources:

Intel MKL Release Notes for the version of the FFTW3 library supported by the wrappers.

www.fftw.org for a description of the MPI FFTW interface.

[Cluster FFT Functions](#) for a description of the Intel MKL CFFT interface.

Building Your Own Wrapper Library

The MPI FFTW wrappers for FFTW3 are delivered as source code, which can be compiled to build a wrapper library.

The source code for the wrappers, makefiles, and files with lists of functions are located in subdirectory `.\interfaces\fftw3x_cdft` in the Intel MKL directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default `INTEGER` type, as well as the name and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the MPI FFTW wrappers to Intel MKL for use from the GNU C compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3x_cdft
make libintel64 compiler=gnu mpi=openmpi INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU `gcc` compiler so that the final executable can use Open MPI, and places the result, named `libfftw3x_cdft_DOUBLE.a`, into directory `/my/path`.

Building an Application

Normally, the only change needed to build your application with MPI FFTW wrappers replacing original FFTW3 library is to add Intel MKL and the wrapper library at the link stage (see section "[Linking Your Application with Intel® Math Kernel Library](#)" in the *Intel MKL User's Guide*).

When you are recompiling your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Running Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW3. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xf_cdft` subdirectory in the Intel MKL directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. Results of running the examples are saved in subdirectory `._results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

See Also

[Building Your Own Wrapper Library](#)

Code Examples

This appendix presents code examples of using some Intel MKL routines and functions.

Please refer to respective chapters in the manual for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors `x` and `y`.

Parameters

<code>n</code>	Specifies the number of elements in vectors <code>x</code> and <code>y</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> .
<code>incy</code>	Specifies the increment for the elements of <code>y</code> .

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5
incx = 2
incy = 1
do i = 1, 10
  x(i) = 2.0e0
  y(i) = 1.0e0
end do
res = sdot (n, x, incx, y, incy)
print*, `SDOT = `, res
end
```

As a result of this program execution, the following line is printed:

```
SDOT = 10.000
```

Example. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector `x` to a vector `y`.

Parameters

<code>n</code>	Specifies the number of elements in vectors <code>x</code> and <code>y</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> .

Parameters

incy Specifies the increment for the elements of *y*.

```

program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
  x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, `Y = `, (y(i), i = 1, n)
end

```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

```
a := alpha*x*y' + a.
```

Parameters

alpha Specifies a scalar *alpha*.

x *m*-element vector.

y *n*-element vector.

a *m*-by-*n* matrix.

```

program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
  x(i) = 1.0
  y(i) = 1.0
end do
do i = 1, m
  do j = 1, n
    a(i,j) = j
  end do
end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, `Matrix A: `
do i = 1, m
  print*, (a(i,j), j = 1, n)
end do
end

```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

```
1.50000 2.50000 3.50000
1.50000 2.50000 3.50000
```

Example. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

```
c := alpha*a*b' + beta*c.
```

Parameters

<i>alpha</i>	Specifies a scalar <i>alpha</i> .
<i>beta</i>	Specifies a scalar <i>beta</i> .
<i>a</i>	Symmetric matrix
<i>b</i>	<i>m</i> -by- <i>n</i> matrix
<i>c</i>	<i>m</i> -by- <i>n</i> matrix

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
  do j = 1, m
    a(i,j) = 1.0
  end do
end do
do i = 1, m
  do j = 1, n
    c(i,j) = 1.0
    b(i,j) = 2.0
  end do
end do
call ssymm (side, uplo, m, n, alpha,
a, lda, b, ldb, beta, c, ldc)
print*, `Matrix C: `
do i = 1, m
  print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

```
5.00000 5.00000
```

5.00000 5.00000

5.00000 5.00000

The following example illustrates a call from a C program to the Fortran version of the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

Fourier Transform Functions Code Examples

This section presents code examples for functions described in the “[FFT Functions](#)” and “[Cluster FFT Functions](#)” sections in the “Fourier Transform Functions” chapter. The examples are grouped in subsections

- [Examples for FFT Functions](#), including [Examples of Using Multi-Threading for FFT Computation](#)
- [Examples for Cluster FFT Functions](#)
- [Auxiliary data transformations](#).

FFT Code Examples

This section presents examples of using the FFT interface functions described in “[Fourier Transform Functions](#)” chapter.

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in “[Configuration Settings](#)”.

In the Fortran examples, the use `mkl_dfti` statement assumes that:

- The `mkl_dfti.f90` module definition file is already compiled.
- The `mkl_dfti.mod` module file is available.

One-dimensional In-place FFT

```
! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
!...put input data into X(1),...,X(32); Y(1),...,Y(32)

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
    DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X(1),X(2),...,X(32)}

! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,&
    DFTI_REAL, 1, 32)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given in CCS format.
```

One-dimensional Out-of-place FFT

```

! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X_in(32)
Complex :: X_out(32)
Real :: Y_in(32)
Real :: Y_out(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X_in(1),...,X_in(32); Y_in(1),...,Y_in(32)
! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
DFTI_COMPLEX, 1, 32 )
Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X_in, X_out )
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by {X_out(1),X_out(2),...,X_out(32)}
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor(My_Desc2_Handle, DFTI_SINGLE,
DFTI_REAL, 1, 32)
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor(My_Desc2_Handle)
Status = DftiComputeForward(My_Desc2_Handle, Y_in, Y_out)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by Y_out in CCS format.

```

Two-dimensional FFT

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```

! Fortran example.
! 2D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X_2D(32,100)
Real :: Y_2D(34, 102)
Complex :: X(3200)
Real :: Y(3468)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status, L(2)
!...put input data into X_2D(j,k), Y_2D(j,k), 1<=j=32,1<=k=100
!...set L(1) = 32, L(2) = 100
!...the transform is a 32-by-100

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,&
DFTI_COMPLEX, 2, L)
Status = DftiCommitDescriptor( My_Desc1_Handle)
Status = DftiComputeForward( My_Desc1_Handle, X)
Status = DftiFreeDescriptor(My_Desc1_Handle)
! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100

```

```
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,&
    DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor(My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100
! and is stored in CCS format
```

The following example demonstrates how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Changing Default Settings

```
! Fortran example
! 1D complex to complex, not in place
Use MKL_DFTI
Complex :: X_in(32), X_out(32)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status
!...put input data into X_in(j), 1<=j<=32
Status = DftiCreateDescriptor( My_Desc_Handle,& DFTI_SINGLE, DFTI_COMPLEX, 1, 32)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)
Status = DftiFreeDescriptor (My_Desc_Handle)
! result is X_out(1),X_out(2),...,X_out(32)
```

Using Status Checking Functions

This example illustrates the use of status checking functions described in [Chapter 11](#).

```
! Fortran
type(DFTI_DESCRIPTOR), POINTER :: desc
integer status
! ...descriptor creation and other code
status = DftiCommitDescriptor(desc)
if (status .ne. 0) then
    if (.not. DftiErrorClass(status,DFTI_NO_ERROR) then
        print *, 'Error: ', DftiErrorMessage(status)
    endif
endif
endif
```

Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```
! Fortran
use mkl_dfti
Complex :: X_2D(20,40)
Complex :: X(800)
Equivalence (X_2D, X)
INTEGER :: STRIDE(2)
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim2
! ...
Status = DftiCreateDescriptor(Desc_Handle_Dim1, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 20 )
Status = DftiCreateDescriptor(Desc_Handle_Dim2, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 40 )
! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )
! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )
```

The following code illustrates real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. [Example "Two-Dimensional REAL In-place FFT \(Fortran Interface\)"](#) is two-dimensional in-place transform and [Example "Two-Dimensional REAL Out-of-place FFT \(Fortran Interface\)"](#) is two-dimensional out-of-place transform in Fortran interface. Note that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Two-Dimensional REAL In-place FFT

```
! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
Real :: X_2D(34,100) ! 34 = (32/2 + 1)*2
Real :: X(3400)
Equivalence (X_2D, X)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_in(3)
Integer :: strides_out(3)
! ...put input data into X_2D(j,k), 1<=j<=32,1<=k<=100
```

```

! ...set L(1) = 32, L(2) = 100
! ...set strides_in(1) = 0, strides_in(2) = 1, strides_in(3) = 34
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17
! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE,&
DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue(My_Desc_Handle, DFTI_INPUT_STRIDES, strides_in)
Status = DftiSetValue(My_Desc_Handle, DFTI_OUTPUT_STRIDES, strides_out)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X )
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in real matrix X_2D in CCE format.

```

Two-Dimensional REAL Out-of-place FFT

```

! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
Real :: X_2D(32,100)
Complex :: Y_2D(17, 100) ! 17 = 32/2 + 1
Real :: X(3200)
Complex :: Y(1700)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_out(3)

! ...put input data into X_2D(j,k), 1<=j<=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17

! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_OUTPUT_STRIDES, strides_out)

Status = DftiCommitDescriptor(My_Desc_Handle)
Status = DftiComputeForward(My_Desc_Handle, X, Y)
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in complex matrix Y_2D in CCE format.

```

Examples of Using OpenMP* Threading for FFT Computation

The following sample program shows how to employ internal OpenMP* threading in Intel MKL for FFT computation.

To specify the number of threads inside Intel MKL, use the following settings:

set MKL_NUM_THREADS = 1 for one-threaded mode;
 set MKL_NUM_THREADS = 4 for multi-threaded mode.

Using Intel MKL Internal Threading Mode (C Example)

```
#include "mkl_dfti.h"

int main ()
{
    float x[200][100];
    DFTI_DESCRIPTOR_HANDLE fft;
    MKL_LONG len[2] = {200, 100};
    // initialize x
    DftiCreateDescriptor ( &fft, DFTI_SINGLE, DFTI_REAL, 2, len );
    DftiCommitDescriptor ( fft );
    DftiComputeForward ( fft, x );
    DftiFreeDescriptor ( &fft );
    return 0;
}
```

The following [Example "Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region"](#) and [Example "Using Parallel Mode with Multiple Descriptors Initialized in One Thread"](#) illustrate a parallel customer program with each descriptor instance used only in a single thread.

Specify the number of OpenMP threads for [Example "Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region"](#) like this:

set MKL_NUM_THREADS = 1 for Intel MKL to work in the single-threaded mode (recommended);
 set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel MKL. To achieve this, you need to set the parameter

DFTI_NUMBER_OF_TRANSFORMS = 4 and to set the corresponding parameter DFTI_INPUT_DISTANCE = 5000.

```
program fft2d_private_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)

  type(dfti_descriptor), pointer :: myFFT
  integer th, myStatus

! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x) PRIVATE(myFFT, myStatus)
  do th = 1, nth
    myStatus = DftiCreateDescriptor (myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
    myStatus = DftiCommitDescriptor (myFFT)
    myStatus = DftiComputeForward (myFFT, x(:, th))
    myStatus = DftiFreeDescriptor (myFFT)
  end do
!$OMP END PARALLEL DO
end
```

Specify the number of OpenMP threads for [Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread”](#) like this:

set MKL_NUM_THREADS = 1 for Intel MKL to work in the single-threaded mode (obligatory);

set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in One Thread

```
program fft2d_array_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)

  type thread_data
    type(dfti_descriptor), pointer :: FFT
  end type thread_data
  type(thread_data) :: workload(nth)

  integer th, status, myStatus

  do th = 1, nth
    status = DftiCreateDescriptor (workload(th)%FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
    status = DftiCommitDescriptor (workload(th)%FFT)
  end do
! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x, workload) PRIVATE(myStatus)
  do th = 1, nth
    myStatus = DftiComputeForward (workload(th)%FFT, x(:, th))
  end do
!$OMP END PARALLEL DO
  do th = 1, nth
    status = DftiFreeDescriptor (workload(th)%FFT)
  end do
end
```

The following [Example “Using Parallel Mode with a Common Descriptor”](#) illustrates a parallel customer program with a common descriptor used in several threads.

Using Parallel Mode with a Common Descriptor

```
program fft2d_shared_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)
  type(dfti_descriptor), pointer :: FFT

  integer th, status, myStatus

  status = DftiCreateDescriptor (FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
  status = DftiCommitDescriptor (FFT)
! assume x is initialized and do 2D FFTs
!$OMP PARALLEL DO SHARED(len, x, FFT) PRIVATE(myStatus)
  do th = 1, nth
    myStatus = DftiComputeForward (FFT, x(:, th))
  end do
```

```
!$OMP END PARALLEL DO
  status = DftiFreeDescriptor (FFT)
end
```

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

2D Out-of-place Cluster FFT Computation

```
DFTI_DESCRIPTOR_DM_HANDLE desc;
MKL_LONG len[2],v,i,j,n,s;
Complex *in,*out;

MPI_Init(...);

// Create descriptor for 2D FFT
len[0]=nx;
len[1]=ny;
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,2,len);
// Ask necessary length of in and out arrays and allocate memory
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
out=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Current process performs n rows,
// 0 row of in corresponds to s row of virtual global array
DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);
// Virtual global array globalIN is defined by function f as
// globalIN[i*ny+j]=f(i,j)
for(i=0;i<n;i++)
  for(j=0;j<ny;j++) in[i*ny+j]=f(i+s,j);
// Set that we want out-of-place transform (default is DFTI_INPLACE)
DftiSetValueDM(desc,DFTI_PLACEMENT,DFTI_NOT_INPLACE);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in,out);
// Virtual global array globalOUT is defined by function g as
// globalOUT[i*ny+j]=g(i,j)
// Now out contains result of FFT. out[i*ny+j]=g(i+s,j)
DftiFreeDescriptorDM(&desc);
free(in);
free(out);
MPI_Finalize();
```

1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```
DFTI_DESCRIPTOR_DM_HANDLE desc;
MKL_LONG len,v,i,n_out,s_out;
Complex *in,*work;

MPI_Init(...);
// Create descriptor for 1D FFT
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,1,len);
// Ask necessary length of array and workspace and allocate memory
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
```

```

work=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Local array has n elements,
// 0 element of in corresponds to s element of virtual global array
DftiGetValueDM(desc,CDFT_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START,&s);
// Set work array as a workspace
DftiSetValueDM(desc,CDFT_WORKSPACE,work);
// Virtual global array globalIN is defined by function f as globalIN[i]=f(i)
for(i=0;i<n;i++) in[i]=f(i+s);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in);
DftiGetValueDM(desc,CDFT_LOCAL_OUT_NX,&n_out);
DftiGetValueDM(desc,CDFT_LOCAL_OUT_X_START,&s_out);
// Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i)
// Now in contains result of FFT. Local array has n_out elements,
// 0 element of in corresponds to s_out element of virtual global array.
// in[i]==g(i+s_out)
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();

```

Auxiliary Data Transformations

This section presents C examples for conversion from the Cartesian to polar representation of complex data and vice versa.

Conversion from Cartesian to polar representation of complex data

```

// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
#include <mkl_vml.h>

void
variant1_Cartesian2Polar(int n,const double *re,const double *im,
                        double *r,double *phi)
{
    vdHypot(n,re,im,r);          // compute radii r[]
    vdAtan2(n,im,re,phi);        // compute phases phi[]
}

void
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                        double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);                // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_im,temp_re,phi); // compute phases phi[]
}

```

Conversion from polar to Cartesian representation of complex data

```
// Polar->Cartesian conversion of complex data.
// Polar representation: z = r * exp( I*phi )
// Cartesian representation: z = re + I*im
#include <mkl_vml.h>

void
variant1_Polar2Cartesian(int n,const double *r,const double *phi,
                        double *re,double *im)
{
    vdSinCos(n,phi,im,re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,re,re); // scale real part
    vdMul(n,r,im,im); // scale imaginary part
}

void
variant2_Polar2Cartesian(int n,const double *r,const double *phi,
                        MKL_Complex16 *z,
                        double *temp_re,double *temp_im)
{
    vdSinCos(n,phi,temp_im,temp_re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,temp_im,temp_im); // scale imaginary part
    vdMul(n,r,temp_re,temp_re); // scale real part
    vdUnpackI(n,temp_re,(double*)z + 0, 2); // fill in result.re
    vdUnpackI(n,temp_im,(double*)z + 1, 2); // fill in result.im
}
```


Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Extended Eigensolver, VM, VS, FFT, and Non-Linear Optimization Solvers functionality, refer to the following publications:

- **BLAS Level 1**

C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

[Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[CXML01] *Compaq Extended Math Library*. Reference Guide, Oct.2001.

[Rem05] K.Remington. *A NIST FORTRAN Sparse Blas User's Guide*. (available on <http://math.nist.gov/~KRemington/fspblas/>)

[Saad94] Y.Saad. *SPARSKIT: A Basic Tool-kit for Sparse Matrix Computation*. Version 2, 1994.(<http://www.cs.umn.edu/~saad>)

[Saad96] Y.Saad. *Iterative Methods for Linear Systems*. PWS Publishing, Boston, 1996.

- **LAPACK**

[AndaPark94] A. A. Anda and H. Park. *Fast plane rotations with dynamic scaling*, SIAM J. matrix Anal. Appl., Vol. 15 (1994), pp. 162-174.

[Baudin12] M. Baudin, R. Smith. *A Robust Complex Division in Scilab*, available from <http://www.arxiv.org>, arXiv:1210.4539v2 (2012).

[Bischof00] C. H. Bischof, B. Lang, and X. Sun. *Algorithm 807: The SBR toolbox-software for successive band reduction*, ACM Transactions on Mathematical Software, Vol. 26, No. 4, pages 602-616, December 2000.

[Demmel92] J. Demmel and K. Veselic. *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl. 13(1992):1204-1246.

[deRijk98] P. P. M. De Rijk. *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Stat. Comp., Vol. 10 (1998), pp. 359-371.

- [Dhillon04] I. Dhillon, B. Parlett. *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- [Dhillon04-02] I. Dhillon, B. Parlett. *Orthogonal Eigenvectors and * Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)
- [Dhillon97] I. Dhillon. *A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.
- [Drmac08-1] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm I*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1322-1342. LAPACK Working note 169.
- [Drmac08-2] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm II*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1343-1362. LAPACK Working note 170.
- [Drmac08-3] Z. Drmac and K. Bujanovic. *On the failure of rank-revealing QR factorization software - a case study*, ACM Trans. Math. Softw. Vol. 35, No 2 (2008), pp. 1-28. LAPACK Working note 176.
- [Drmac08-4] Z. Drmac. *Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic*, SIAM J. Sci. Comp., Vol. 18 (1997), pp. 1200-1222.
- [Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition, 1996.
- [LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.
- [Kahan66] W. Kahan. *Accurate Eigenvalues of a Symmetric Tridiagonal Matrix*, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.
- [Marques06] O. Marques, E.J. Riedy, and Ch. Voemel. *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM Journal on Scientific Computing, Vol. 28, No. 5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)
- [Sutton09] Brian D. Sutton. *Computing the complete CS decomposition*, Numer. Algorithms, 50(1):33-65, 2009.
- **ScaLAPACK**

[SLUG] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.
 - **Sparse Solver**

[Duff99] I. S. Duff and J. Koster. *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices*. SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.

- [Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. *Reverse Communication Interface for Linear Algebra Templates for Iterative Methods*. UT-CS-95-291, May 1995. <http://www.netlib.org/lapack/lawnspdf/lawn99.pdf>
- [Karypis98] G. Karypis and V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. SIAM Journal on Scientific Computing, 20(1): 359-392, 1998.
- [Li99] X.S. Li and J.W. Demmel. *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.
- [Liu85] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.
- [Menon98] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46-55, 1998. <http://www.openmp.org>.
- [Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, SIAM, Philadelphia, PA, 2003.
- [Schenk00] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000.
- [Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1): 158-176, 2000.
- [Schenk01] O. Schenk and K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARDISO*. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.
- [Schenk02] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28:187-197, 2002.
- [Schenk03] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.
- [Schenk04] O. Schenk and K. Gartner. *On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems*. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.
- [Sonn89] P. Sonneveld. *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.
- [Young71] D.M.Young. *Iterative Solution of Large Linear Systems*. New York, Academic Press, Inc., 1971.
- **Extended Eigensolver**
- [Polizzi09] E. Polizzi, *Density-Matrix-Based Algorithms for Solving Eigenvalue Problems*, Phys. Rev. B. Vol. 79, 115112, 2009.
- [Polizzi12] E. Polizzi, *A High-Performance Numerical Library for Solving Eigenvalue Problems: FEAST Solver v2.0 User's Guide*, arxiv.org/abs/1203.4031, 2012.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe and H. van der Vorst, editors, *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[Sleijpen96] G. L. G. Sleijpen and H. A. van der Vorst. *A Jacobi-Davidson iteration method for linear eigenvalue problems*. SIAM J. Matrix Anal. Appl., 17:401-425, 1996.

• **VS**

[AVX] Intel. *Intel® Advanced Vector Extensions Programming Reference*. (<http://software.intel.com/file/36945>)

[Billor00] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. *BACON: blocked adaptive computationally efficient outlier nominators*. Computational Statistics & Data Analysis, 34, 279-298, 2000.

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.

[Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[BMT] Intel. *Bull Mountain Technology Software Implementation Guide*. (<http://software.intel.com/file/37157>)

[Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.

[Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.

[IntelSWMan] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 3 vols. (<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>)

[L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77-120, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.

[L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.

[Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.

[Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[Maronna02] Maronna, R.A., and Zamar, R.H., *Robust Multivariate Estimates for High-Dimensional Datasets*, Technometrics, 44, 307-317, 2002.

- [Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.
- [Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, <http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html>.
- [NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp
- [Rocke96] David M. Rocke, *Robustness properties of S-estimators of multivariate location and shape in high dimension*. The Annals of Statistics, 24(3), 1327-1345, 1996.
- [Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.
<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html>
- [Salmon11] Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.
- [Schafer97] Schafer, J.L., *Analysis of Incomplete Multivariate Data*. Chapman & Hall, 1997.
- [Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).
- [SSL Notes] *Intel® MKL Summary Statistics Application Notes*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- [VS Notes] *Intel® MKL Vector Statistics Notes*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- [VS Data] *Intel® MKL Vector Statistics Performance*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- **VM**
- [C99] ISO/IEC 9899:1999/Cor 3:2007. Programming languages -- C.
- [Muller97] J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.
- [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.
- [VM Data] *Intel® MKL Vector Mathematics Performance and Accuracy*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- **FFT**
- [1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.

- [2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.
- [3] Ping Tak Peter Tang, *DFTI - a new interface for Fast Fourier Transform libraries*, ACM Transactions on Mathematical Software, Vol. 31, Issue 4, Pages 475 - 507, 2005.
- [4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

- **Optimization Solvers**

- [Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.
- [Dong95] J. Dongarra, V. Eijkhout, A. Kalhan. *Reverse communication interface for linear algebra templates for iterative methods*. 1995.

- **Data Fitting Functions**

- [deBoor2001] Carl deBoor. *A Practical Guide to Splines*. Revised Edition. Springer-Verlag New York Berlin Heidelberg, 2001.
- [Schumaker2007] Larry L Schumaker. *Spline Functions: Basic Theory*. 3rd Edition. Cambridge University Press, Cambridge, 2007.
- [StechSub76] S.B. Stechhkin, and Yu Subbotin. *Splines in Numerical Mathematics*. Izd. Nauka, Moscow, 1976.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages visit www.netlib.org.



Glossary

A^H	Denotes the conjugate transpose of a general matrix A . See also conjugate matrix.
A^T	Denotes the transpose of a general matrix A . See also transpose.
band matrix	A general m -by- n matrix A such that $a_{ij} = 0$ for $ i - j > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and <i>diagonals</i> of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^H P^T$ (or $A = PLDL^H P^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .
c	When found as the first letter of routine names, c indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable X . For univariate distribution the cumulative distribution function is the function of real argument x , which for every x takes a value equal to probability of the event $A: X \leq x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \dots, x_n)$, which, for every x , takes a value equal to probability of the event $A = (X_1 \leq x_1 \ \& \ X_2 \leq x_2, \ \& \ \dots, \ \& \ X_n \leq x_n)$.

Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ or $A = L L^H$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \ A\ \ A^{-1}\ $.
conjugate matrix	The matrix A^H defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$.
d	When found as the first letter of routine names, d indicates the usage of double-precision real data type.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \sum_i x_i y_i$. Here x_i and y_i stand for the i -th elements of x and y , respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23 \times 10^{-308} < x < 1.79 \times 10^{308}$. For this data type, the machine precision ϵ is approximately 10^{-15} , which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
eigenvalue	See eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix A and the vectors x are called the eigenvectors of the matrix A .
eigenvector	See eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau v v^T$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book.
full storage	A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
Hermitian matrix	A square matrix A that is equal to its conjugate matrix A^H . The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.
I	See identity matrix.
identity matrix	A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A , $AI = A$ and $IA = A$.

i.i.d.	Independent Identically Distributed.
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.
Intel MKL	Abbreviation for Intel® Math Kernel Library.
inverse matrix	The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A .
LQ factorization	Representation of an m -by- n matrix A as $A = LQ$ or $A = (L \ 0)Q$. Here Q is an n -by- n orthogonal (unitary) matrix. For $m \leq n$, L is an m -by- m lower triangular matrix with real diagonal elements; for $m > n$, where L_1 is an n -by- n lower triangular matrix, and L_2 is a rectangular matrix.
LU factorization	Representation of a general m -by- n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).
machine precision	The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See also double precision and single precision.
MPI	Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.
MPICH	A freely available, portable implementation of MPI standard for message-passing libraries.
orthogonal matrix	A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^T A = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.
packed storage	A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.
PDF	Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X . The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$. For univariate distribution the relation is

$$F(x) = \int_{-\infty}^x f(t)dt .$$

For multivariate distribution the relation is

$$F(x_1, x_2, \dots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \dots \int_{-\infty}^{x_n} f(t_1, t_2, \dots, t_n) dt_1 dt_2 \dots dt_n$$

positive-definite matrix	A square matrix A such that $Ax \cdot x > 0$ for any non-zero vector x . Here \cdot denotes the dot product.
pseudorandom number generator	A completely deterministic algorithm that imitates truly random sequences.
QR factorization	Representation of an m -by- n matrix A as $A = QR$, where Q is an m -by- m orthogonal (unitary) matrix, and R is n -by- n upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.
random stream	An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.
RNG	Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.
Rectangular Full Packed (RFP) storage	A storage scheme combining the full and packed storage schemes for the upper or lower triangle of the matrix. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.
s	When found as the first letter of routine names, s indicates the usage of single-precision real data type.
ScaLAPACK	Stands for Scalable Linear Algebra PACKage.
Schur factorization	Representation of a square matrix A in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex A , triangular matrix) called the Schur form of A ; the matrix Z is orthogonal (for complex A , unitary). Columns of Z are called Schur vectors.
single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18 \times 10^{-38} < x < 3.40 \times 10^{38}$. For this data type, the machine precision (ϵ) is approximately 10^{-7} , which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H . See also SVD.

SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. See BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. See full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. See <i>also</i> Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.
transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T , and columns of A become rows of A^T).
trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^H A = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. See Chapter 9 of this book.
VSL	Abbreviation for Vector Statistical Library. See Chapter 10 of this book.
z	When found as the first letter of routine names, z indicates the usage of double-precision complex data type.

?gesc21320
?gesd2d3054
?gesdd1164
?gesv646
?gesvd1160
?gesvj1174
?gesvx650
?gesvxx656
?getc21322
?getf21323
?getrf425
?getri599
?getrs465
?ggbak994
?ggbal992
?gges1254
?ggesx1260
?ggev1267
?ggevxx1271
?ggglm1060
?ggghrd989
?gglse1058
?ggqrf830
?ggrqf833
?ggsvd1178
?ggsvp1023
?gsum2d3051
?gsvj01688
?gsvj11690
?gtcon508
?gthr160
?gthrz161
?gtrfs552
?gtsv680
?gtsvx682
?gttrf432
?gttrs469
?gtts21324
?hbev1111
?hbevd1116
?hbevxx1124
?hbgst939
?hbgv1237
?hbgvd1242
?hbgvx1250
?hbtrd900
?hecon521
?hecon_rook523
?heequb644
?heev1066
?heevd1071
?heevr1087
?heevx1078
?heft21671, 1673
?hegst932
?hegv1196
?hegvd1202
?hegvx1210
?hemm129
?hemv88
?her90
?her292
?her2k137
?herdb875
?herfs579
?herfsx581
?herk135
?hesv737
?hesv_rook740
?hesvx742
?hesvxx746
?heswapr1662
?hetrd881
?hetrf453
?hetrf_rook456
?hetri609
?hetri_rook611
?hetri2614
?hetri2x618
?hetrs485
?hetrs_rook487
?hetrs2491
?hfrk1695
?hgeqz996
?hpcon526
?hpev1094
?hpevd1099
?hpevxx1106
?hpgst935
?hpgv1217
?hpgvd1223
?hpgvx1230
?hpmv93
?hpr95
?hpr297
?hprfs589
?hpsv759
?hpsvx761
?hptrd893
?hptrf461
?hptri621
?hptrs494
?hsein966
?hseqr962
?isnan1325
?jacobi2971
?jacobi_delete2970
?jacobi_init2968
?jacobi_solve2969
?jacobix2972
?la_gbamv1719
?la_gbrcond1721
?la_gbrcond_c1723
?la_gbrcond_x1725
?la_gbrfsx_extended1726
?la_gbrpvgwr1733
?la_geamv1734
?la_gercond1736
?la_gercond_c1737
?la_gercond_x1739
?la_gerfsx_extended1740
?la_gerpvgwr1775
?la_heamv1746
?la_hercond_c1748
?la_hercond_x1749
?la_herfsx_extended1751
?la_herpvgwr1757
?la_porcond1759
?la_porcond_c1760
?la_porcond_x1762
?la_porfsx_extended1763
?la_porpvgwr1770
?la_syamv1686, 1778
?la_syrcond1780
?la_syrcond_c1781
?la_syrcond_x1783
?la_syrfsx_extended1784
?la_syrpvgwr1791
?la_wwaddw1792
?labrd1326

?lacgv1291
?lacn21329
?lacon1330
?laczp21718
?lacpy1332
?lacrm1292
?lactr1293
?ladiv1333
?lae21334
?laebz1335
?laed01339
?laed11341
?laed21343
?laed31345
?laed41347
?laed51348
?laed61349
?laed71350
?laed81354
?laed91357
?laeda1358
?laein1360
?laesy1294
?laev21362
?laexc1364
?lag21365
?lags21367
?lagtf1370
?lagtm1372
?lagts1373
?lagv21375
?lahef1575
?lahef_rook1577
?lahqr1377
?lahr21381
?lahrd1379
?laic11384
?laisnan1326
?lain21387
?lals01390
?lalsa1393
?lalsd1396
?lamc11805
?lamc21805
?lamc31806
?lamc41807
?lamc51808
?lamch1804
?lamrg1398
?lamsh2278
?laneg1399
?langb1400
?lange1402
?langt1403
?lanhb1407
?lanhe1413
?lanhf1700
?lanhp1410
?lanhs1404
?lansb1405
?lansf1698
?lansp1408
?lanst/?lanht1411
?lansy1412
?lantb1415
?lantp1416
?lantr1418
?lanv21420
?lapll1421
?lapmr1422
?lapmt1423
?lapy21424
?lapy31425
?laqgb1425
?laqge1427
?laqhb1429
?laqhe1771
?laqhp1772
?laqp21430
?laqps1432
?laqr01434
?laqr11437
?laqr21438
?laqr31442
?laqr41445
?laqr51448
?laqsb1451
?laqsp1453
?laqsy1454
?laqtr1456
?lar1v1458
?lar2v1461
?larcm1774
?laref2285
?larf1463
?larfb1464
?larfg1468
?larfgp1469
?larfp1685
?larft1471
?larfx1473
?largv1476
?larnv1478
?larra1485
?larrb1487
?larrc1489
?larrd1490
?larre1493
?larrr1497
?larrj1499
?larrrk1501
?larrrr1502
?larrrv1503
?larscl21776
?lartg1507
?lartgp1508
?lartgs1510
?lartv1511
?laruv1512
?larz1513
?larzb1515
?larzt1517
?las21520
?lascl1520
?lascl21777
?lasd01522
?lasd11523
?lasd21526
?lasd31529
?lasd41532
?lasd51533
?lasd61534
?lasd71538
?lasd81542
?lasd91544
?lasda1546
?lasdq1549
?lasdt1551
?laset1552
?lasorte2310

?lasq11553
?lasq21555
?lasq31556
?lasq41557
?lasq51559
?lasq61560
?lasr1561
?lasrt1564
?lasrt22311
?lassq1565
?lasv21567
?laswp1568
?lasy21569
?lasyf1571
?lasyf_rook1573
?latbs1579
?latdf1608
?latms1813
?latps1609
?latrd1612
?latrs1614
?latrz1618
?lauu21620
?lauum1621
?nrm263
?opgtr890
?opmtr891
?orbdb/?unbdb1039
?orbdb1/?unbdb11622
?orbdb2/?unbdb21625
?orbdb3/?unbdb31628
?orbdb4/?unbdb41631
?orbdb5/?unbdb51635
?orbdb6/?unbdb61637
?orcsd/?uncsd1184
?orcsd2by1/?orcsd2by11188
?org2l/?ung2l1640
?org2r/?ung2r1641
?orgbr849
?orgbr948
?orgl2/?ungl21642
?orglq793
?orgql803
?orgqr781
?orgr2/?ungr21644
?orgrq814
?orgtr877
?orm2l/?unm2l1645
?orm2r/?unm2r1647
?ormbr852
?ormhr950
?orml2/?unml21649
?ormlq795
?ormql807
?ormqr783
?ormr2/?unmr21651
?ormr3/?unmr31653
?ormrq817
?ormrz825
?ormtr879
?pbcon514
?pbequ640
?pbrfs565
?pbstf941
?pbsv709
?pbsvx711
?pbt21655
?pbtrf443
?pbtrs478
?pftf439
?pftri603
?pfttrs474
?pocon510
?poequ635
?poequb637
?porfs555
?porfsx557
?posv688
?posvx691
?posvxx696
?potf21657
?potrf436
?potri601
?potrs472
?ppcon512
?ppequ638
?pprfs563
?ppsv703
?ppsvx705
?pptrf441
?pptri604
?pptrs476
?pstf21714
?pstrf438
?ptcon515
?pteqr919
?ptrfs568
?ptsv716
?ptsvx718
?pttrf444
?pttrs480
?pttrsv2331
?ptts21658
?rot64, 1295
?rotg65
?roti162
?rotm66
?rotmg68
?rscl1659
?sbev1109
?sbevd1113
?sbevxx1120
?sbgst937
?sbgv1234
?sbgvd1239
?sbgvx1246
?sbmv99
?sbtrd898
?scal69
?sctr164
?sdot59
?sfrk1693
?spcon524
?spev1092
?spevd1096
?spevx1102
?spgst934
?spgv1215
?spgvd1219
?spgvx1226
?spmv101, 1296
?spr103, 1298
?spr2105
?sprfs587
?spsv754
?spsvx756
?sptrd888
?sptrf458
?sptri619
?sptrs492

?stebz922
 ?stedc911
 ?stegr915
 ?stein925
 ?stemr907
 ?steqr904
 ?steqr22332
 ?sterf903
 ?stev1128
 ?stevd1129
 ?stevr1136
 ?stevx1133
 ?sum11304
 ?swap71
 ?sycon517
 ?sycon_rook519
 ?syconv1299
 ?syequb642
 ?syev1064
 ?syevd1068
 ?syevr1082
 ?syevx1074
 ?sygs2/?hegs21665
 ?sygst930
 ?sygv1193
 ?sygvd1199
 ?sygvx1205
 ?symm140
 ?symv107, 1301
 ?syr109, 1302
 ?syr2110
 ?syr2k145
 ?syrd872
 ?syrf570
 ?syrfx573
 ?syrk143
 ?sysv721
 ?sysv_rook723
 ?sysvx726
 ?sysvxx730
 ?syswapr1660
 ?syswapr11663
 ?sytd2/?hetd21666
 ?sytf21668
 ?sytf2_rook1670
 ?sytrd870
 ?sytrf446
 ?sytrf_rook450
 ?sytri606
 ?sytri_rook607
 ?sytri2612
 ?sytri2x616
 ?sytrs482
 ?sytrs_rook484
 ?sytrs2489
 ?tbcon532
 ?tbmv112
 ?tbsv114
 ?tbtrs501
 ?tfsm1696
 ?tftri624
 ?tfttp1701
 ?tfttr1702
 ?tgevc1002
 ?tgex21674
 ?tgexc1006
 ?tgsen1009
 ?tgsja1027
 ?tgsna1019
 ?tgsy21677
 ?tgsyl1015
 ?tpcon530
 ?tpmqrt839
 ?tpmv117
 ?tpqrt836
 ?tpqrt21704
 ?tprfb1706
 ?tprfs594
 ?tpsv119
 ?tptri625
 ?tptrs499
 ?tpddf1709
 ?tpdtr1711
 ?trbr2d3060
 ?trbs2d3058
 ?trcon528
 ?trevc971
 ?trexc980
 ?trmm148
 ?trmv121
 ?trnlsp_check2955
 ?trnlsp_delete2959
 ?trnlsp_get2958
 ?trnlsp_init2953
 ?trnlsp_solve2956
 ?trnlspbc_check2962
 ?trnlspbc_delete2967
 ?trnlspbc_get2966
 ?trnlspbc_init2960
 ?trnlspbc_solve2965
 ?trrfs592
 ?trrv2d3056
 ?trsd2d3054
 ?trsen982
 ?trsm151
 ?trsna975
 ?trsv123
 ?trsvl986
 ?trti21680
 ?trtri (LAPACK)623
 ?trtrs (LAPACK)496
 ?trttf1712
 ?trttp1713
 ?tzrzf822
 ?ungbr855
 ?unghr953
 ?unqlq797
 ?unql805
 ?ungqr785
 ?ungrq815
 ?ungtr884
 ?unmbr858
 ?unmhr955
 ?unmlq799
 ?unmq1809
 ?unmqr788
 ?unmrq820
 ?unmrz827
 ?unmtr885
 ?upgtr895
 ?upmtr896
 1-norm value
 complex Hermitian matrix
 packed storage1410
 complex Hermitian matrix in RFP format1700
 complex Hermitian tridiagonal matrix1411
 complex symmetric matrix1412
 general rectangular matrix1402, 2148

general tridiagonal matrix1403
 Hermitian band matrix1407
 real symmetric matrix1412, 2151
 real symmetric matrix in RFP format1698
 real symmetric tridiagonal matrix1411
 symmetric band matrix1405
 symmetric matrix
 packed storage1408
 trapezoidal matrix1418
 triangular band matrix1415
 triangular matrix
 packed storage1416
 upper Hessenberg matrix1404, 2149

A

absolute value of a vector element
 largest72
 smallest73
 accuracy modes, in VM2483
 adding magnitudes of elements of a distributed
 vector2819
 adding magnitudes of the vector elements54
 arguments
 matrix3145
 sparse vector154
 vector3143
 array descriptor1819, 2815
 auxiliary functions
 ?la_lin_berr1758
 auxiliary routines
 ScaLAPACK2089

B

backward error1758
 balancing a matrix958
 band storage scheme3145
 basic quasi-number generator
 Niederreiter2599
 Sobol2599
 basic random number generators
 GFSR2599
 MCG, 32-bit2599
 MCG, 59-bit2599
 Mersenne Twister
 MT199372599
 MT22032599
 MRG2599
 Wichmann-Hill2599
 bdlaapp2095
 bdlaexc2096
 bdsdc864
 bdtrexc2098
 Bernoulli2663
 Beta2654
 bidiagonal matrix
 LAPACK842
 ScaLAPACK1986
 Binomial2666
 bisection1487
 BLACS
 broadcast3056
 combines3047
 destruction routines3067
 informational routines3069
 initialization routines3061
 miscellaneous routines3071
 point to point communication3052

?gamn2d3049
 ?gamx2d3048
 ?gebr2d3059
 ?gebs2d3058
 ?gerv2d3055
 ?gesd2d3054
 ?gsum2d3051
 ?trbr2d3060
 ?trbs2d3058
 ?trrv2d3056
 ?trsd2d3054
 blacs_abort3068
 blacs_barrier3071
 blacs_exit3069
 blacs_freebuff3068
 blacs_get3062
 blacs_gridexit3068
 blacs_gridinfo3070
 blacs_gridinit3065
 blacs_gridmap3066
 blacs_pcoord3070
 blacs_pinfo3061
 blacs_pnum3070
 blacs_set3063
 blacs_setup3062
 usage examples3072
 BLACS routines
 coherence3044
 matrix shapes3043
 repeatability3044
 blacs_abort3068
 blacs_barrier3071
 blacs_exit3069
 blacs_freebuff3068
 blacs_get3062
 blacs_gridexit3068
 blacs_gridinfo3070
 blacs_gridinit3065
 blacs_gridmap3066
 blacs_pcoord3070
 blacs_pinfo3061
 blacs_pnum3070
 blacs_set3063
 blacs_setup3062
 BLAS Code Examples3163
 BLAS Level 1 routines
 ?asum54
 ?axpby388
 ?axpy54, 56
 ?cabs154, 74
 ?copy54, 57
 ?dot54, 58
 ?dotc54, 60
 ?dotu54, 61
 ?nrm254, 63
 ?rot54, 64
 ?rotg54, 65
 ?rotm54, 66
 ?rotmg68
 ?rotmq54
 ?scal54, 69
 ?sdot54, 59
 ?swap54, 71
 code example3163
 i?amax54, 72
 i?amin54, 73
 BLAS Level 2 routines
 ?gbmv74, 75
 ?gem2vc391
 ?gem2vu389

- ?gemv74, 78
 - ?ger74, 81
 - ?gerc74, 82
 - ?geru74, 84
 - ?hbm74, 85
 - ?hemv74, 88
 - ?her74, 90
 - ?her274, 92
 - ?hpmv74, 93
 - ?hpr74, 95
 - ?hpr274, 97
 - ?sbmv74, 99
 - ?spm74, 101
 - ?spr74, 103
 - ?spr274, 105
 - ?symv74, 107
 - ?syr74, 109
 - ?syr274, 110
 - ?t74, 112
 - ?tbsv74, 114
 - ?tpmv74, 117
 - ?tpsv74, 119
 - ?trmv74, 121
 - ?trsv74, 123
 - code example3164
 - BLAS Level 3 routines
 - ?gemm125, 126
 - ?gemm_batch131
 - ?gemm3m397
 - ?gemm3m_batch400
 - ?hemm125, 129
 - ?her2k125, 137
 - ?herk125, 135
 - ?symm125, 140
 - ?syr2k125, 145
 - ?syrk125, 143
 - ?tfsm1696
 - ?trmm125, 148
 - ?trsm125, 151
 - code example3165
 - BLAS routines
 - routine groups51
 - BLAS-like extensions387
 - BLAS-like transposition routines
 - mkl_?imatcopy404
 - mkl_?omatadd412
 - mkl_?omatcopy407
 - mkl_?omatcopy2409
 - block reflector
 - general matrix
 - LAPACK1515
 - ScaLAPACK2197
 - general rectangular matrix
 - LAPACK1464
 - ScaLAPACK2183
 - triangular factor
 - LAPACK1471, 1517
 - ScaLAPACK2191, 2203
 - block-cyclic distribution1819, 2815
 - block-splitting method2599
 - BRNG2593, 2594, 2599
 - bslaapp2095
 - bslaexc2096
 - bstrexc2098
 - Bunch-Kaufman factorization
 - Hermitian matrix
 - packed storage461
 - symmetric matrix
 - packed storage458
- ## C
- Cauchy2643
 - cbbcsd1034
 - cgbcon506
 - cgbrfsx545
 - cgbsvx666
 - cgbtrs467
 - cgecon504
 - cgemmt394
 - cgeqpf776
 - cgtrfs552
 - chegs21665
 - cheswapr1662
 - chetd21666
 - chetri2614
 - chetri2x618
 - chetrs2491
 - chgeqz996
 - chla_transtype1808
 - Cholesky factorization
 - Hermitian positive semi-definite matrix1714
 - Hermitian positive semidefinite matrix438
 - Hermitian positive-definite matrix
 - band storage443, 478, 711, 1833, 1848
 - packed storage441, 705
 - split941
 - symmetric positive semi-definite matrix1714
 - symmetric positive semidefinite matrix438
 - symmetric positive-definite matrix
 - band storage443, 478, 711, 1833, 1848
 - packed storage441, 705
 - chseqr962
 - cla_gbamv1719
 - cla_gbrcond_c1723
 - cla_gbrcond_x1725
 - cla_gbrfsx_extended1726
 - cla_gbrpvgrw1733
 - cla_geamv1734
 - cla_gercond_c1737
 - cla_gercond_x1739
 - cla_gerfsx_extended1740
 - cla_gerpvgrw1775
 - cla_heamv1746
 - cla_hercond_c1748
 - cla_hercond_x1749
 - cla_herfsx_extended1751
 - cla_herpvgrw1757
 - cla_lin_berr1758
 - cla_porcond_c1760
 - cla_porcond_x1762
 - cla_porfsx_extended1763
 - cla_porpvgrw1770
 - cla_syamv1778
 - cla_syrcond_c1781
 - cla_syrcond_x1783
 - cla_syrfsx_extended1784
 - cla_syrpvgrw1791
 - cla_wwaddw1792
 - clag2z1681
 - clakf21386
 - clapmr1422
 - clapmt1423
 - clarfb1464
 - clarft1471
 - clarge1475
 - clarnd1477
 - claror1479
 - clarot1482
 - clarscl21776
 - clascl21777

clatm11581
 clatm21583
 clatm31586
 clatm51589
 clatm61593
 clatme1596
 clatmr1600
 clatps1609
 clatrd1612
 clatrs1614
 clatrz1618
 clauu21620
 clauum1621
 cluster_sparse_solver2385
 cluster_sparse_solver iparm parameter2391
 CNR, support functions for
 mkl_cbwr_get3032
 mkl_cbwr_get_auto_branch3033
 mkl_cbwr_set3031
 code examples
 BLAS Level 1 function3163
 BLAS Level 1 routine3163
 BLAS Level 2 routine3164
 BLAS Level 3 routine3165
 communication subprograms1819
 complex division in real arithmetic1333
 complex Hermitian matrix
 1-norm value
 LAPACK1413
 ScaLAPACK2151
 factorization with diagonal pivoting method1671,
 1673
 Frobenius norm
 LAPACK1413
 ScaLAPACK2151
 infinity- norm
 LAPACK1413
 ScaLAPACK2151
 largest absolute value of element
 LAPACK1413
 ScaLAPACK2151
 complex Hermitian matrix in packed form
 1-norm value1410
 Frobenius norm1410
 infinity- norm1410
 largest absolute value of element1410
 complex Hermitian tridiagonal matrix
 1-norm value1411
 Frobenius norm1411
 infinity- norm1411
 largest absolute value of element1411
 complex matrix
 complex elementary reflector
 ScaLAPACK2201
 complex symmetric matrix
 1-norm value1412
 Frobenius norm1412
 infinity- norm1412
 largest absolute value of element1412
 complex vector
 1-norm using true absolute value
 LAPACK1304
 ScaLAPACK2105
 conjugation
 LAPACK1291
 ScaLAPACK2100
 complex vector conjugation
 LAPACK1291
 ScaLAPACK2100
 component-wise relative error1758
 compressed sparse vectors154
 computational node2595
 Computational Routines765
 condition number
 band matrix506
 general matrix
 LAPACK504
 ScaLAPACK1855, 1857, 1860
 Hermitian matrix
 packed storage526
 Hermitian positive-definite matrix
 band storage514
 packed storage512
 tridiagonal515
 symmetric matrix
 packed storage524
 symmetric positive-definite matrix
 band storage514
 packed storage512
 tridiagonal515
 triangular matrix
 band storage532
 packed storage530
 tridiagonal matrix508
 configuration parameters, in FFT interface2761
 Configuration Settings, for Fourier transform
 functions2761, 2795
 Continuous Distribution Generators2626
 Continuous Distributions2629
 ConvCopyTask2705
 ConvDeleteTask2704
 converting a DOUBLE COMPLEX triangular matrix to
 COMPLEX1717
 converting a double-precision triangular matrix to single-
 precision1716
 converting a sparse vector into compressed storage form
 and writing zeros to the original vector161
 converting compressed sparse vectors into full storage
 form164
 ConvInternalPrecision2692
 Convolution and Correlation2676
 Convolution Functions
 ?ConvExec2695
 ?ConvExec1D2697
 ?ConvExecX2699
 ?ConvExecX1D2702
 ?ConvNewTask2682
 ?ConvNewTask1D2684
 ?ConvNewTaskX2685
 ?ConvNewTaskX1D2688
 ConvCopyTask2705
 ConvDeleteTask2704
 ConvSetDecimation2694
 ConvSetInternalPrecision2692
 ConvSetMode2691
 ConvSetStart2693
 CorrCopyTask2705
 CorrDeleteTask2704
 ConvSetMode2691
 ConvSetStart2693
 copying
 distributed vectors2822
 matrices
 distributed2136
 global parallel2138
 local replicated2138
 two-dimensional
 LAPACK1332, 1718
 ScaLAPACK2139
 vectors57

copying a matrix1701, 1702, 1709, 1711–1713
 CopyStream2615
 CopyStreamState2616
 CorrCopyTask2705
 CorrDeleteTask2704
 Correlation Functions
 ?CorrExec2695
 ?CorrExec1D2697
 ?CorrExecX2699
 ?CorrExecX1D2702
 ?CorrNewTask2682
 ?CorrNewTask1D2684
 ?CorrNewTaskX2685
 ?CorrNewTaskX1D2688
 CorrSetDecimation2694
 CorrSetInternalPrecision2692
 CorrSetMode2691
 CorrSetStart2693
 CorrSetInternalDecimation2694
 CorrSetInternalPrecision2692
 CorrSetMode2691
 CorrSetStart2693
 cosine-sine decomposition
 LAPACK1033, 1183
 cpbtf21655
 cporfsx557
 cpotf21657
 cprfs563
 cptrs476
 cptts21658
 Cray2334
 crscl1659
 cs decomposition
 See also LAPACK routines, cs decomposition1033
 CSD (cosine-sine decomposition)
 LAPACK1033, 1183
 csyconv1299
 csyswapr1660
 csyswapr1663
 csytf21668
 csytf2_rook1670
 csytri2612
 csytri2x616
 csytrs2489
 ctgex21674
 ctgsy21677
 ctrex980
 ctri21680
 cunbdb1039
 cunbdb11622
 cunbdb21625
 cunbdb31628
 cunbdb41631
 cunbdb51635
 cunbdb61637
 cuncsd1184
 cuncsd2by11188
 cunm2l1640
 cunm2r1641
 cunmbr855
 cunmgl21642
 cunm2l1644
 cunm2l1645
 cunm2r1647
 cunml21649
 cunmr21651
 cunmr31653

D

data type
 in VM2483
 shorthand43
 Data Types2602
 dbbcscd1034
 dbdsdc864
 dcg_check2427
 dcg_get2429
 dcg_init2426
 dcgmrhs_check2431
 dcgmrhs_get2434
 dcgmrhs_init2430
 DeleteStream2614
 descriptor configuration
 cluster FFT2803
 descriptor manipulation
 cluster FFT2803
 DF task3083
 dfdconstruct1d3107
 dfdeditdxptr3102
 dfdeditppspline1d3092
 dfdeditptr3099
 dfdeletetask3123
 dfdintegrate1d3113
 dfdintegrateex1d3113
 dfdintegrallback3120
 dfdinterpcallback3119
 dfdinterpolate1d3108
 dfdinterpolateex1d3108
 dfdnewtask1d3090
 dfdqueryidxptr3105
 dfdqueryptr3103
 dfdsearchcells1d3117
 dfdsearchcellscallback3121
 dfdsearchcellsex1d3117
 dfgmres_check2436
 dfgmres_get2439
 dfgmres_init2435
 dfieditval3100
 dfiqueryval3104
 dfsconstruct1d3107
 dfseditdxptr3102
 dfseditppspline1d3092
 dfseditptr3099
 dfsintegrate1d3113
 dfsintegrateex1d3113
 dfsintegrallback3120
 dfsinterpcallback3119
 dfsinterpolate1d3108
 dfsinterpolateex1d3108
 dfsnewtask1d3090
 dfsqueryidxptr3105
 dfsqueryptr3103
 dfssearchcells1d3117
 dfssearchcellscallback3121
 dfssearchcellsex1d3117
 DFT routines
 descriptor configuration
 DftiSetValue2786
 DftiCommitDescriptor2783
 DftiCommitDescriptorDM2805
 DftiComputeBackward2792
 DftiComputeBackwardDM2808
 DftiComputeForward2790
 DftiComputeForwardDM2806
 DftiCopyDescriptor2785
 DftiCreateDescriptor2781
 DftiCreateDescriptorDM2804
 DftiErrorClass2797

DftiErrorMessage2799
 DftiFreeDescriptor2784
 DftiFreeDescriptorDM2805
 DftiGetValue2788
 DftiGetValueDM2812
 DftiSetValue2786
 DftiSetValueDM2810
 dgbcon506
 dgbrfsx545
 dgbsvx666
 dgbtrs467
 dgecon504
 dgejsv1167
 dgemmt394
 dgeqp776
 dgesvj1174
 dgtrfs552
 dhgeqz996
 dhseqr962
 diagonal elements
 LAPACK1552
 ScaLAPACK2208
 diagonal pivoting factorization
 Hermitian indefinite matrix746
 symmetric indefinite matrix730
 diagonally dominant tridiagonal matrix
 solving systems of linear equations471
 diagonally dominant-like banded matrix
 solving systems of linear equations1842
 diagonally dominant-like tridiagonal matrix
 solving systems of linear equations1845
 dimension3143
 Direct Sparse Solver (DSS) Interface Routines2395
 Discrete Distribution Generators2627
 Discrete Distributions2657
 Discrete Fourier Transform
 DftiSetValue2786
 distributed complex matrix
 transposition2885, 2886
 distributed general matrix
 matrix-vector product2831, 2834
 rank-1 update2836
 rank-1 update, unconjugated2840
 rank-l update, conjugated2838
 distributed Hermitian matrix
 matrix-vector product2842, 2844
 rank-1 update2846
 rank-2 update2847
 rank-k update2872
 distributed matrix equation
 $AX = B$ 2890
 distributed matrix-matrix operation
 rank-k update
 distributed Hermitian matrix2872
 transposition
 complex matrix2885
 complex matrix, conjugated2886
 real matrix2884
 distributed matrix-vector operation
 product
 Hermitian matrix2842, 2844
 symmetric matrix2849, 2851
 triangular matrix2857, 2859
 rank-1 update
 Hermitian matrix2846
 symmetric matrix2853
 rank-1 update, conjugated2838
 rank-1 update, unconjugated2840
 rank-2 update
 Hermitian matrix2847
 symmetric matrix2855
 distributed real matrix
 transposition2884
 distributed symmetric matrix
 matrix-vector product2849, 2851
 rank-1 update2853
 rank-2 update2855
 distributed triangular matrix
 matrix-vector product2857, 2859
 solving systems of linear equations2861
 distributed vector-scalar product2828
 distributed vectors
 adding magnitudes of vector elements2819
 copying2822
 dot product
 complex vectors2826
 complex vectors, conjugated2825
 real vectors2823
 Euclidean norm2827
 global index of maximum element2818
 linear combination of vectors2820
 sum of vectors2820
 swapping2829
 vector-scalar product2828
 distributed-memory computations1819
 Distribution Generators2626
 Distribution Generators Supporting Accurate Mode2627, 2628
 divide and conquer algorithm2036, 2053
 djacobi2971
 djacobi_delete2970
 djacobi_init2968
 djacobi_solve2969
 djacobix2972
 dla_gbamv1719
 dla_gbrcond1721
 dla_gbrfsx_extended1726
 dla_gbrpvgrw1733
 dla_geamv1734
 dla_gercond1736
 dla_gerfsx_extended1740
 dla_gerpvgrw1775
 dla_lin_berr1758
 dla_porcond1759
 dla_porfsx_extended1763
 dla_porpvgrw1770
 dla_syamv1778
 dla_syrcond1780
 dla_syrfsx_extended1784
 dla_syrpvgrw1791
 dla_wwaddw1792
 dlag2s1682
 dlakf21386
 dlapmr1422
 dlapmt1423
 dlaqr62279
 dlar1va2282
 dlaran1462
 dlarfb1464
 dlarft1471
 dlarge1475
 dlarnd1477
 dlaror1479
 dlarot1482
 dlarrb22287
 dlarrd22290
 dlarre22293
 dlarre2a2298
 dlarrf22302
 dlarrv22304

dlarscl21776
 dlartgp1508
 dlartgs1510
 dlascl21777
 dlat2s1716
 dlatm11581
 dlatm21583
 dlatm31586
 dlatm51589
 dlatm61593
 dlatme1596
 dlatmr1600
 dlatps1609
 dlatrd1612
 dlatrs1614
 dlatrz1618
 dlauu21620
 dlauum1621
 dNewAbstractStream2611
 dorbdb1039
 dorbdb11622
 dorbdb21625
 dorbdb31628
 dorbdb41631
 dorbdb51635
 dorbdb61637
 dorcsd1184
 dorcsd2by11188
 dorg2l1640
 dorg2r1641
 dorgl21642
 dorgr21644
 dorm2l1645
 dorm2r1647
 dorml21649
 dormr21651
 dormr31653
 dot product
 complex vectors, conjugated60
 complex vectors, unconjugated61
 distributed complex vectors, conjugated2825
 distributed complex vectors, unconjugated2826
 distributed real vectors2823
 real vectors58
 real vectors (double precision)59
 sparse complex vectors159
 sparse complex vectors, conjugated158
 sparse real vectors157
 dpbtf21655
 dporfsx557
 dpotf21657
 dpprfs563
 dpptrs476
 dptts21658
 driver
 expert1821
 simple1821
 Driver Routines646, 1044
 drscl1659
 dss_create2399
 dstegr22312
 dstegr2a2315
 dstegr2b2319
 dsyconv1299
 dsygs21665
 dsyswapr1660
 dsyswapr11663
 dsytd21666
 dsytf21668
 dsytf2_rook1670
 dsytri2612
 dsytri2x616
 dsytrs2489
 dtgex21674
 dtgsy21677
 dtrexc980
 dtrnisp_check2955
 dtrnisp_delete2959
 dtrnisp_get2958
 dtrnisp_init2953
 dtrnisp_solve2956
 dtrnispbc_check2962
 dtrnispbc_delete2967
 dtrnispbc_get2966
 dtrnispbc_init2960
 dtrnispbc_solve2965
 dtrti21680
 dzsum11304

E
 eigenpairs, sorting2310
 eigenvalue problems
 general matrix943, 989, 1971
 generalized form929
 Hermitian matrix867
 symmetric matrix867
 symmetric tridiagonal matrix2324, 2332
 eigenvalues
 bisection2287
 eigenvalue problems867
 eigenvalues and eigenvectors2315
 eigenvectors
 eigenvalue problems867
 scaled2282
 elementary reflector
 complex matrix2201
 general matrix1513, 2194
 general rectangular matrix
 LAPACK1463, 1473
 ScaLAPACK2180, 2187
 LAPACK generation1468, 1469
 ScaLAPACK generation2190
 Enter index keyword337–339, 358, 370
 error diagnostics, in VM2487
 error estimation for linear equations
 distributed tridiagonal coefficient matrix1870
 error handling
 pxerbla2989
 xerbla51, 2487, 2815
 error handling for fatal errors2990
 errors in solutions of linear equations
 banded matrix545, 1726, 1763
 distributed tridiagonal coefficient matrix1870
 general matrix
 band storage543
 Hermitian indefinite matrix581, 1751
 Hermitian matrix
 packed storage589
 Hermitian positive-definite matrix
 band storage565
 packed storage563
 symmetric indefinite matrix573, 1784
 symmetric matrix
 packed storage587
 symmetric positive-definite matrix
 band storage565
 packed storage563
 triangular matrix

- band storage597
- packed storage594
- tridiagonal matrix552
- Estimates3106
- Euclidean norm
 - of a distributed vector2827
 - of a vector63
- exception handling2988
- expert driver1821
- Exponential2637
- Extended Eigensolver2453
- Extended Eigensolver interface
 - ?feast_hbev2471
 - ?feast_hbgv2473
 - ?feast_hcsrev2476
 - ?feast_hcsrgv2478
 - ?feast_heev2466
 - ?feast_hegv2468
 - ?feast_hrci2462
 - ?feast_sbev2471
 - ?feast_sbgv2473
 - ?feast_scsrev2476
 - ?feast_scsrgv2478
 - ?feast_srcv2462
 - ?feast_syev2466
 - ?feast_sygv2468
- feastinit2457
- Extended Eigensolvers2455–2460, 2462, 2465, 2466, 2468, 2471, 2473, 2476, 2478

F

- factorization
 - Bunch-Kaufman
 - LAPACK424
 - ScaLAPACK1823
 - Cholesky
 - LAPACK424, 1655, 1657
 - ScaLAPACK2257
 - diagonal pivoting
 - Hermitian matrix
 - complex1671, 1673
 - packed761
 - symmetric matrix
 - indefinite1668, 1670
 - packed756
 - LU
 - LAPACK424
 - ScaLAPACK1823
 - orthogonal
 - LAPACK766
 - ScaLAPACK1882
 - partial
 - complex Hermitian indefinite matrix1575, 1577
 - real/complex symmetric matrix1571, 1573
 - triangular factorization424, 1823
 - upper trapezoidal matrix1618
- fast Fourier transform
 - DftiCommitDescriptor2783
 - DftiCommitDescriptorDM2805
 - DftiComputeBackward2792
 - DftiComputeBackwardDM2808
 - DftiComputeForwardDM2806
 - DftiCopyDescriptor2785
 - DftiCreateDescriptor2781
 - DftiCreateDescriptorDM2804
 - DftiErrorClass2797
 - DftiErrorMessage2799

- DftiFreeDescriptor2784
- DftiFreeDescriptorDM2805
- DftiGetValue2788
- DftiGetValueDM2812
- DftiSetValueDM2810
- fast Fourier Transform
 - DftiComputeForward2790
- fatal error, handling2990
- feastinit2457
- FFT computation
 - cluster FFT2803
- FFT functions
 - descriptor manipulation
 - DftiCommitDescriptor2783
 - DftiCommitDescriptorDM2805
 - DftiCopyDescriptor2785
 - DftiCreateDescriptor2781
 - DftiCreateDescriptorDM2804
 - DftiFreeDescriptor2784
 - DftiFreeDescriptorDM2805
 - DFT computation
 - DftiComputeBackward2792
 - DftiComputeForward2790
 - FFT computation
 - DftiComputeForwardDM2806
 - status checking
 - DftiErrorClass2797
 - DftiErrorMessage2799
- FFT Interface2761
- FFT routines
 - descriptor configuration
 - DftiGetValue2788
 - DftiGetValueDM2812
 - DftiSetValueDM2810
 - FFT computation
 - DftiComputeBackwardDM2808
- FFTW interface to Intel(R) MKL
 - for FFTW23151
 - for FFTW33157
- fill-in, for sparse matrices3127
- finding
 - index of the element of a vector with the largest absolute value of the real part2101
 - element of a vector with the largest absolute value72
 - element of a vector with the largest absolute value of the real part and its global index2104
 - element of a vector with the smallest absolute value73
- font conventions43
- Fortran 95 interface conventions
 - BLAS, Sparse BLAS53
 - LAPACK418
- Fortran 95 LAPACK interface vs. Netlib419
- free_Helmholtz_2D2926
- free_Helmholtz_3D2926
- free_sph_np2933
- free_sph_p2933
- free_trig_transform2904
- Frobenius norm
 - complex Hermitian matrix
 - packed storage1410
 - complex Hermitian matrix in RFP format1700
 - complex Hermitian tridiagonal matrix1411
 - complex symmetric matrix1412
 - general rectangular matrix1402, 2148
 - general tridiagonal matrix1403
 - Hermitian band matrix1407
 - real symmetric matrix1412, 2151
 - real symmetric matrix in RFP format1698
 - real symmetric tridiagonal matrix1411

- symmetric band matrix1405
 - symmetric matrix
 - packed storage1408
 - trapezoidal matrix1418
 - triangular band matrix1415
 - triangular matrix
 - packed storage1416
 - upper Hessenberg matrix1404, 2149
 - full storage scheme3145
 - full-storage vectors154
 - function name conventions, in VM2484
- G**
- Gamma2652
 - gathering sparse vector's elements into compressed form
 - and writing zeros to these elements161
 - Gaussian2632
 - GaussianMV2634
 - gbcon506
 - gbsvx666
 - gbtrs467
 - gecon504
 - general distributed matrix
 - scalar-matrix-matrix product2868
 - general matrix
 - block reflector1515, 2197
 - copying2338
 - eigenvalue problems943, 989, 1971
 - elementary reflector1513, 2194
 - estimating the condition number
 - band storage506
 - inverting matrix
 - LAPACK599
 - ScaLAPACK1873
 - LQ factorization790, 1897
 - LU factorization
 - band storage430, 1305, 1825, 1827, 2326, 2327
 - matrix-vector product
 - band storage75
 - multiplying by orthogonal matrix
 - from LQ factorization1649, 2243
 - from QR factorization1647, 2239
 - from RQ factorization1651, 2246
 - from RZ factorization1653
 - multiplying by unitary matrix
 - from LQ factorization1649, 2243
 - from QR factorization1647, 2239
 - from RQ factorization1651, 2246
 - from RZ factorization1653
 - QL factorization
 - LAPACK801
 - ScaLAPACK1909
 - QR factorization
 - with pivoting776, 779, 1885
 - rank-1 update81
 - rank-1 update, conjugated82
 - rank-1 update, unconjugated84
 - reduction to bidiagonal form1306, 1326, 2114
 - reduction to upper Hessenberg form2117
 - RQ factorization
 - LAPACK811
 - ScaLAPACK1946
 - scalar-matrix-matrix product126, 131, 397, 400
 - Schur factorization reordering2098, 2268
 - solving systems of linear equations
 - band storage
 - LAPACK467
 - ScaLAPACK1840
 - general rectangular distributed matrix
 - computing scaling factors1878
 - equilibration1878
 - general rectangular matrix
 - 1-norm value
 - LAPACK1402
 - ScaLAPACK2148
 - block reflector
 - LAPACK1464
 - ScaLAPACK2183
 - elementary reflector
 - LAPACK1463, 2187
 - ScaLAPACK2180
 - Frobenius norm
 - LAPACK1402
 - ScaLAPACK2148
 - infinity- norm
 - LAPACK1402
 - ScaLAPACK2148
 - largest absolute value of element
 - LAPACK1402
 - ScaLAPACK2148
 - LQ factorization
 - LAPACK1310
 - ScaLAPACK2119
 - multiplication
 - LAPACK1520
 - ScaLAPACK2206
 - QL factorization
 - LAPACK1311
 - ScaLAPACK2122
 - QR factorization
 - LAPACK1313, 1314
 - ScaLAPACK2124
 - reduction of first columns
 - LAPACK1379, 1381
 - ScaLAPACK2142
 - reduction to bidiagonal form2130
 - row interchanges
 - LAPACK1568
 - ScaLAPACK2212
 - RQ factorization
 - LAPACK1319
 - ScaLAPACK1921, 2126
 - scaling2157
 - general square matrix
 - reduction to upper Hessenberg form1308
 - trace2214
 - general triangular matrix
 - LU factorization
 - band storage2106
 - general tridiagonal matrix
 - 1-norm value1403
 - Frobenius norm1403
 - infinity- norm1403
 - largest absolute value of element1403
 - general tridiagonal triangular matrix
 - LU factorization
 - band storage2108
 - generalized eigenvalue problems
 - complex Hermitian positive-definite problem
 - band storage939
 - packed storage935
 - complex Hermitian-definite problem1666, 2264
 - real symmetric-definite problem
 - band storage937
 - packed storage934
 - See also LAPACK routines, generalized eigenvalue problems929

Generalized LLS Problems1057
 Generalized Nonsymmetric Eigenproblems1254
 generalized Schur factorization1375, 1461, 1476, 1478
 Generalized Singular Value Decomposition1023
 generalized Sylvester equation1015
 Generalized SymmetricDefinite Eigenproblems1193
 generation methods2594
 Geometric2664
 geqpf776
 GetBrngProperties2676
 GetNumRegBrngs2625
 GetStreamSize2621
 GetStreamStateBrng2625
 GFSR2596
 Givens rotation
 modified Givens transformation parameters68
 of sparse vectors162
 parameters65
 global array2815
 global index of maximum element of a distributed
 vector2818
 global matrix1819
 gtrfs552
 Gumbel2650

H

Helmholtz problem
 three-dimensional2914
 two-dimensional2911
 Helmholtz problem on a sphere
 non-periodic2912
 periodic2912
 Hermitian band matrix
 1-norm value1407
 Frobenius norm1407
 infinity- norm1407
 largest absolute value of element1407
 Hermitian distributed matrix
 rank-n update2874
 scalar-matrix-matrix product2870
 Hermitian indefinite matrix
 matrix-vector product1746
 Hermitian matrix
 Bunch-Kaufman factorization
 packed storage461
 eigenvalues and eigenvectors2050, 2053, 2055,
 2061, 2103
 estimating the condition number
 packed storage526
 generalized eigenvalue problems929
 inverting the matrix
 packed storage621
 matrix-vector product
 band storage85
 packed storage93
 rank-1 update
 packed storage95
 rank-2 update
 packed storage97
 rank-2k update137
 rank-k update135
 reducing to standard form
 LAPACK1665
 ScaLAPACK2262
 reducing to tridiagonal form
 LAPACK1612, 1666
 ScaLAPACK2215, 2264
 scalar-matrix-matrix product129

scaling2176
 solving systems of linear equations
 packed storage494
 Hermitian positive definite distributed matrix
 computing scaling factors1880
 equilibration1880
 Hermitian positive semidefinite matrix
 Cholesky factorization438
 Hermitian positive-definite band matrix
 Cholesky factorization1655
 Hermitian positive-definite distributed matrix
 inverting the matrix1875
 Hermitian positive-definite matrix
 Cholesky factorization
 band storage443, 1833
 packed storage441
 estimating the condition number
 band storage514
 packed storage512
 inverting the matrix
 packed storage604
 solving systems of linear equations
 band storage478, 1848
 packed storage476
 Hermitian positive-definite tridiagonal matrix
 solving systems of linear equations1851
 Hessenberg matrix
 eigenvalues2160
 eigenvalues and eigenvectors2171
 eigenvalues and Schur factorization1983
 heswapr1662
 hetri2614
 hetri2x618
 hgeqz996
 Householder matrix
 LAPACK1468, 1469
 ScaLAPACK2190
 Householder reflector2285
 hseqr962
 Hypergeometric2668

I

i?amax72
 i?amin73
 i?max11304
 IBM ESSL library2677
 IEEE arithmetic2145
 IEEE standard
 implementation2335
 signbit position2337
 ila?lr1687
 iladiag1809
 ilaenv1798
 ilaprec1810
 ilatrans1810
 ilauplo1811
 ilaver1798
 ILU0 preconditioner2440
 Incomplete LU Factorization Technique2440
 increment3143
 iNewAbstractStream2609
 infinity-norm
 complex Hermitian matrix
 packed storage1410
 complex Hermitian matrix in RFP format1700
 complex Hermitian tridiagonal matrix1411
 complex symmetric matrix1412
 general rectangular matrix1402, 2148

- general tridiagonal matrix1403
- Hermitian band matrix1407
- real symmetric matrix1412, 2151
- real symmetric matrix in RFP format1698
- real symmetric tridiagonal matrix1411
- symmetric band matrix1405
 - symmetric matrix
 - packed storage1408
- trapezoidal matrix1418
- triangular band matrix1415
 - triangular matrix
 - packed storage1416
- upper Hessenberg matrix1404, 2149
- Intel(R) Many Integrated Core Architecture support
 - functions
 - mkl_mic_clear_status3027
 - mkl_mic_disable3010
 - mkl_mic_enable3010
 - mkl_mic_free_memory3016
 - mkl_mic_get_cpuinfo3029
 - mkl_mic_get_device_count3011
 - mkl_mic_get_flags3024
 - mkl_mic_get_meminfo3028
 - mkl_mic_get_resource_limit3022
 - mkl_mic_get_status3025
 - mkl_mic_get_workdivision3013
 - mkl_mic_register_memory3017
 - mkl_mic_set_device_num_threads3018
 - mkl_mic_set_flags3024
 - mkl_mic_set_max_memory3015
 - mkl_mic_set_offload_report3023
 - mkl_mic_set_resource_limit3019
 - mkl_mic_set_workdivision3012
 - mkl_set_env_mode3039
- Interface Consideration168
- inverse matrix. inverting a matrix599, 1873, 1875, 1877
- inverting a matrix
 - general matrix
 - LAPACK599
 - ScaLAPACK1873
 - Hermitian matrix
 - packed storage621
 - Hermitian positive-definite matrix
 - LAPACK601
 - packed storage604
 - ScaLAPACK1875
 - symmetric matrix
 - packed storage619
 - symmetric positive-definite matrix
 - LAPACK601
 - packed storage604
 - ScaLAPACK1875
 - triangular distributed matrix1877
 - triangular matrix
 - packed storage625
- iparmq1800
- Iterative Sparse Solvers2411
- Iterative Sparse Solvers based on Reverse Communication
 - Interface (RCI ISS)2411
- J**
- Jacobi plane rotations1174
- Jacobian matrix calculation routines
 - ?jacobi2971
 - ?jacobi_delete2970
 - ?jacobi_init2968
 - ?jacobi_solve2969
 - ?jacobix2972
- L**
- la_gbamv1719
- la_gbrcond1721
- la_gbrcond_c1723
- la_gbrcond_x1725
- la_gercond1736
- la_gercond_c1737
- la_gercond_x1739
- la_hercond_c1748
- la_hercond_x1749
- la_lin_berr1758
- la_porcond1759
- la_porcond_c1760
- la_porcond_x1762
- la_syrcond1780
- la_syrcond_c1781
- la_syrcond_x1783
- LAPACK
 - naming conventions417
- LAPACK auxiliary routines
 - ?la_geamv1734
 - ?la_heamv1746
 - ?la_syamv1778
 - ?larscl21776
 - ?lascl21777
- LAPACK routines
 - ?gsvj01688
 - ?gsvj11690
 - ?hfrk1695
 - ?larfp1685
 - ?sfrk1693
- 2-by-2 generalized eigenvalue problem1365
 - 2-by-2 Hermitian matrix
 - plane rotation1461
- 2-by-2 orthogonal matrices1367
 - 2-by-2 real matrix
 - generalized Schur factorization1375
 - 2-by-2 real nonsymmetric matrix
 - Schur factorization1420
 - 2-by-2 symmetric matrix
 - plane rotation1461
 - 2-by-2 triangular matrix
 - singular values1520
 - SVD1567
- approximation to smallest eigenvalue1557
 - auxiliary routines
 - ?gbtf21305
 - ?geb21306
 - ?gehd21308
 - ?gelq21310
 - ?geql21311
 - ?geqr21313
 - ?geqr2p1314
 - ?gerq21319
 - ?gesc21320
 - ?getc21322
 - ?getf21323
 - ?gtts21324
 - ?hetf21671
 - ?hetf2_rook1673
 - ?hfrk1695
 - ?isnan1325
 - ?la_gbrpvgrw1733
 - ?la_gerpvgrw1775
 - ?la_herpvgrw1757
 - ?la_porpvgrw1770
 - ?la_syrpvgrw1791
 - ?la_wwaddw1792
 - ?labrd1326
 - ?lacgv1291
 - ?lacn21329
 - ?lacon1330

?lacp21718	?laqr01434
?lacpy1332	?laqr11437
?lacrm1292	?laqr21438
?lact1293	?laqr31442
?ladiv1333	?laqr41445
?lae21334	?laqr51448
?laebz1335	?laqsb1451
?laed01339	?laqsp1453
?laed11341	?laqsy1454
?laed21343	?laqtr1456
?laed31345	?lar1v1458
?laed41347	?lar2v1461
?laed51348	?larcm1774
?laed61349	?larf1463
?laed71350	?larfb1464
?laed81354	?larfg1468
?laed91357	?larfgp1469
?laeda1358	?larfp1685
?laein1360	?larft1471
?laesy1294	?larfx1473
?laev21362	?largv1476
?laexc1364	?larlv1478
?lag21365	?larra1485
?lags21367	?larrb1487
?lagtf1370	?larrc1489
?lagtm1372	?larrrd1490
?lagts1373	?larre1493
?lagv21375	?larrf1497
?lahef1575	?larrj1499
?lahef_rook1577	?larrk1501
?lahqr1377	?larrl1502
?lahr21381	?larrv1503
?lahrd1379	?lartg1507
?laic11384	?lartgp1508
?laisnan1326	?lartgs1510
?laln21387	?lartv1511
?lals01390	?laruv1512
?lalsa1393	?larz1513
?lalsd1396	?larzb1515
?lamrg1398	?larzt1517
?laneg1399	?las21520
?langb1400	?lascl1520
?lange1402	?lasd01522
?langt1403	?lasd11523
?lanhb1407	?lasd21526
?lanhe1413	?lasd31529
?lanhf1700	?lasd41532
?lanhp1410	?lasd51533
?lanhs1404	?lasd61534
?lansb1405	?lasd71538
?lansf1698	?lasd81542
?lansp1408	?lasd91544
?lanst/?lanht1411	?lasda1546
?lansy1412	?lasdq1549
?lantb1415	?lasdt1551
?lantp1416	?laset1552
?lantr1418	?lasq11553
?lanv21420	?lasq21555
?lapll1421	?lasq31556
?lapmr1422	?lasq41557
?lapmt1423	?lasq51559
?lapy21424	?lasq61560
?lapy31425	?lasr1561
?laqgb1425	?lasrt1564
?laqge1427	?lassq1565
?laqhb1429	?lasv21567
?laqhe1771	?laswp1568
?laqhp1772	?lasy21569
?laqp21430	?lasyf1571
?laqps1432	?lasyf_rook1573

?latbs1579
 ?latdf1608
 ?latps1609
 ?latrd1612
 ?latrs1614
 ?latrz1618
 ?lauu21620
 ?lauum1621
 ?orbdb/?
 unbdb
 1039
 ?orbdb1/?
 unbdb1
 1622
 ?orbdb2/?
 unbdb2
 1625
 ?orbdb3/?
 unbdb3
 1628
 ?orbdb4/?
 unbdb4
 1631
 ?orbdb5/?
 unbdb5
 1635
 ?orbdb6/?
 unbdb6
 1637
 ?orcsd/?
 uncsd
 1184
 ?orcsd2by1/?
 uncsd2by
 11188
 ?org2l/?
 ung2l
 1640
 ?org2r/?
 ung2r
 1641
 ?orgl2l/?
 ungl2
 1642
 ?orgr2/?
 ungr2
 1644
 ?orm2l/?
 unm2l
 1645
 ?orm2r/?
 unm2r
 1647
 ?orml2/?
 unml2
 1649
 ?ormr2/?
 unmr2
 1651
 ?ormr3/?
 unmr3
 1653
 ?pbt21655
 ?potf21657
 ?pstf21714
 ?ptts21658
 ?rot1295
 ?rscl1659
 ?sfrk1693
 ?spmv1296
 ?spr1298
 ?sum11304
 ?sygs2/?
 hegs2
 1665
 ?symv1301
 ?syr1302
 ?sytd2/?
 hetd2
 1666
 ?sytf21668
 ?sytf2_rook1670
 ?tfttp1701
 ?tfttr1702
 ?tgex21674
 ?tgsy21677
 ?tpttf1709
 ?tpttr1711
 ?trti21680
 ?trttf1712
 ?trttp1713
 clag2z1681
 dlag2s1682
 dlat2s1716
 i?max11304
 ila?lc1686
 ila?lr1687
 slag2d1683
 zlag2c1684
 zlat2c1717
 banded matrix equilibration
 ?gbequ631
 ?gbequ633
 bidiagonal divide and conquer1551
 block reflector
 triangular factor1471, 1517
 checking for safe infinity1802
 complex Hermitian matrix
 packed storage1410
 complex Hermitian matrix in RFP
 format1700
 complex Hermitian tridiagonal matrix1411
 complex matrix multiplication1292, 1774
 complex symmetric matrix
 computing eigenvalues and
 eigenvectors1294
 matrix-vector product1301
 symmetric rank-1 update1302
 complex symmetric packed matrix
 symmetric rank-1 update1298
 complex vector
 1-norm using true absolute
 value1304
 index of element with max absolute
 value1304
 linear transformation1293
 matrix-vector product1296
 plane rotation1295
 complex vector conjugation1291
 condition number estimation
 ?disna928
 ?gbcon506
 ?gecon504
 ?gtcon508
 ?hecon521
 ?hecon_rook523
 ?hpcon526
 ?pbcon514
 ?pocon510
 ?ppcon512

?ptcon515	?gbsvx666
?spcon524	?gbsvxx672
?sycon517	?gesv646
?sycon_rook519	?gesvx650
?tbcon532	?gesvxx656
?tpcon530	?gtsv680
?trcon528	?gtsvx682
determining machine parameters1805	?hesv737
diagonally dominant triangular	?hesv_rook740
factorization	?hesvx742
?dttrfb434	?hesvxx746
dqd transform1560	?hpsv759
dqds transform1559	?hpsvx761
driver routines	?pbsv709
generalized LLS problems	?pbsvx711
?ggglm1060	?posv688
?gglse1058	?posvx691
generalized nonsymmetric	?posvxx696
eigenproblems	?ppsv703
?gges1254	?ppsvx705
?ggesx1260	?ptsv716
?ggev1267	?ptsvx718
?ggev1271	?spsv754
generalized symmetric	?spsvx756
definite	?sysv721
eigenproblems	?sysv_rook723
?hbgv1237	?sysvx726
?hbgvd1242	?sysvxx730
?hbgvx1250	symmetric eigenproblems
?hegv1196	?hbev1111
?hegvd1202	?hbevd1116
?hegvx1210	?hbev1124
?hpgv1217	?heev1066
?hpgvd1223	?heevd1071
?hpgvx1230	?heevr1087
?sbgv1234	?heevx1078
?sbgvd1239	?hpev1094
?sbgvx1246	?hpevd1099
?spgv1215	?hpevx1106
?spgvd1219	?sbev1109
?spgvx1226	?sbevd1113
?sygv1193	?sbev1120
?sygvd1199	?spev1092
?sygvx1205	?spevd1096
linear least squares	?spevx1102
problems	?stev1128
?gels1045	?stevd1129
?gelsd1054	?stevr1136
?gels1051	?stevx1133
?gelsy1048	?syev1064
?lals0 (auxiliary)1390	?syevd1068
?lalsd (auxiliary)1393	?syevr1082
?lalsd (auxiliary)1396	?syevx1074
nonsymmetric	environmental enquiry1798, 1800
eigenproblems	finding a relatively isolated eigenvalue1497
?gees1141	general band matrix
?geesx1145	equilibration1425
?geev1150	general matrix
?geevx1154	?latms1813
singular value	block reflector1515
decomposition	elementary reflector1513
?gejsv1167	reduction to bidiagonal form1306,
?gelsd1054	1326
?gesdd1164	general matrix equilibration
?gesvd1160	?geequ627
?gesvj1174	?geequb629
?ggsvd1178	general rectangular matrix
solving linear equations	block reflector1464
?dtsvb686	elementary reflector1463, 1473
?gbsv664	equilibration1427, 1771, 1772

- LQ factorization1310
- plane rotation1561
- QL factorization1311
- QR factorization1313, 1314
- row interchanges1568
- RQ factorization1319
- general square matrix
 - reduction to upper Hessenberg form1308
- general tridiagonal matrix1370, 1372, 1373, 1403, 1493, 1503
- generalized eigenvalue problems
 - ?hbgst939
 - ?hegst932
 - ?hpgst935
 - ?pbstf941
 - ?sbgst937
 - ?spgst934
 - ?sygst930
- generalized SVD
 - ?ggsvp1023
 - ?tgsja1027
- generalized Sylvester equation
 - ?tgsyl1015
- Hermitian band matrix
 - equilibration1429, 1454
- Hermitian band matrix in packed storage
 - equilibration1453
- Hermitian indefinite matrix equilibration
 - ?heequ644
- Hermitian matrix
 - computing eigenvalues and eigenvectors1362
- Hermitian positive-definite matrix equilibration
 - ?poequ635
 - ?poequ637
- Householder matrix
 - elementary reflector1468, 1469
- ila?lc1686
- ila?lr1687
- incremental condition estimation1384
- linear dependence of vectors1421
 - LQ factorization
 - ?gelq21310
 - ?gelqf790
 - ?orglq793
 - ?ormlq795
 - ?unglq797
 - ?unmlq799
 - LU factorization
 - general band matrix1305
 - matrix equilibration
 - ?laqgb1425
 - ?laqge1427
 - ?laqhb1429
 - ?laqhe1771
 - ?laqhp1772
 - ?laqsb1451
 - ?laqsp1453
 - ?laqsy1454
 - ?pbequ640
 - ?ppequ638
 - matrix inversion
 - ?getri599
 - ?hetri609
 - ?hetri_rook611
 - ?hetri2614
 - ?hetri2x618
- ?hptri621
- ?potri601
- ?pptri604
- ?sptri619
- ?sytri606
- ?sytri_rook607
- ?sytri2612
- ?sytri2x616
- ?tptri625
- ?trtri623
- matrix-matrix product
 - ?lagtm1372
- merging sets of singular values1526, 1538
- mixed precision iterative refinement
 - subroutines646, 688, 1681–1684
- nonsymmetric eigenvalue problems
 - ?gebak960
 - ?gebal958
 - ?gehrd946
 - ?hsein966
 - ?hseqr962
 - ?orghr948
 - ?ormhr950
 - ?trevc971
 - ?trexc980
 - ?trsen982
 - ?trсна975
 - ?unghr953
 - ?unmhr955
- off-diagonal and diagonal elements1552
- permutation list creation1398
- permutation of matrix columns1423
- permutation of matrix rows1422
- plane rotation1507, 1508, 1510, 1511, 1561
- plane rotation vector1476
 - QL factorization
 - ?geql21311
 - ?geqlf801
 - ?orgql803
 - ?ormql807
 - ?ungql805
 - ?unmql809
 - QR factorization
 - ?gemqrt774
 - ?geqp3779
 - ?geqpf776
 - ?geqr21313
 - ?geqr2p1314
 - ?geqrf767
 - ?geqrfp770
 - ?geqrt772
 - ?geqrt21316
 - ?geqrt31317
 - ?ggqrf830
 - ?ggrqf833
 - ?laqp21430
 - ?laqps1432
 - ?orgqr781
 - ?ormqr783
 - ?tpmqrt839
 - ?tpqrt836
 - ?tpqrt21704
 - ?tprfb1706
 - ?ungqr785
 - ?unmqr788
 - p?geqrf1883
- random numbers vector1478
 - real lower bidiagonal matrix SVD1549
 - real square bidiagonal matrix

singular values1553
 real symmetric matrix1412
 real symmetric matrix in RFP format1698
 real symmetric tridiagonal matrix1335, 1411
 real upper bidiagonal matrix
 singular values1522
 SVD1523, 1546, 1549
 real upper quasi-triangular matrix
 orthogonal similarity
 transformation1364
 reciprocal condition numbers for
 eigenvalues and/or
 eigenvectors
 ?tgsna1019
 rectangular full packed format439, 474
 RQ factorization
 ?geqr21319
 ?gerqf811
 ?orgqr814
 ?ormrq817
 ?ungrq815
 ?unmrq820
 RZ factorization
 ?ormrz825
 ?tzrzf822
 ?unmrz827
 singular value decomposition
 ?bdsdc864
 ?bdsqr861
 ?gbbbrd846
 ?gebrd843
 ?orgbr849
 ?ormbr852
 ?ungbr855
 ?unmbr858
 solution refinement and error
 estimation
 ?gbrfs543
 ?gbrfsx545
 ?gerfs534
 ?gerfsx536
 ?gtrfs552
 ?herfs579
 ?herfsx581
 ?hprfs589
 ?la_gbrfsx_extended1726
 ?la_gerfsx_extended1740
 ?la_herfsx_extended1751
 ?la_porfsx_extended1763
 ?la_syrfxs_extended1784
 ?pbrfs565
 ?porfs555
 ?porfsx557
 ?pprfs563
 ?ptrfs568
 ?sprfs587
 ?syrfs570
 ?syrfsx573
 ?tbrfs597
 ?tprfs594
 ?trrfs592
 solving linear equations
 ?dttrs471
 ?gbtrs467
 ?getrs465
 ?gttrs469
 ?heswapr1662
 ?hetrs485
 ?hetrs_rook487
 ?hetrs2491
 ?hptrs494
 ?laln21387
 ?laqtr1456
 ?pbtrs478
 ?pfttrs474
 ?potrs472
 ?pptrs476
 ?pttrs480
 ?sptrs492
 ?syswapr1660
 ?syswapr1663
 ?sytrs482
 ?sytrs_rook484
 ?sytrs2489
 ?tbtrs501
 ?tptrs499
 ?trtrs496
 sorting numbers1564
 square root1424, 1425
 square roots1529, 1532, 1533, 1542, 1544,
 1803
 Sylvester equation
 ?lasy21569
 ?tgsy21677
 ?trsy1986
 symmetric band matrix
 equilibration1451, 1454
 symmetric band matrix in packed
 storage
 equilibration1453
 symmetric eigenvalue problems
 ?disna928
 ?hbtrd900
 ?herdb875
 ?hetrd881
 ?hptrd893
 ?opgtr890
 ?opmtr891
 ?orgtr877
 ?ormtr879
 ?pteqr919
 ?sbtrd898
 ?sptrd888
 ?stebz922
 ?stedc911
 ?stegr915
 ?stein925
 ?stemr907
 ?steqr904
 ?sterf903
 ?syrdb872
 ?sytrd870
 ?ungtr884
 ?unmtr885
 ?upgtr895
 ?upmtr896
 auxiliary
 ?lae21334
 ?laebz1335
 ?laed01339
 ?laed11341
 ?laed21343
 ?laed31345
 ?laed41347
 ?laed51348
 ?laed61349
 ?laed71350
 ?laed81354
 ?laed91357
 ?laeda1358

- symmetric indefinite matrix
 - equilibration
 - ?sysequb642
- symmetric matrix
 - computing eigenvalues and eigenvectors1362
 - packed storage1408
- symmetric positive-definite matrix
 - equilibration
 - ?poequ635
 - ?poequb637
- symmetric positive-definite tridiagonal matrix
 - eigenvalues1555
- test routines
 - ?latms1813
- trapezoidal matrix1418, 1618
- triangular factorization
 - ?gbtrf430
 - ?getrf425
 - ?gttrf432
 - ?hetrf453
 - ?hetrf_rook456
 - ?hptrf461
 - ?pbtrf443
 - ?potrf436
 - ?pptrf441
 - ?pstrf438
 - ?pttrf444
 - ?sptrf458
 - ?sytrf446
 - ?sytrf_rook450
 - mkl_?spffrt2463
 - mkl_?spffrtx463
 - p?dbtrf1827
- triangular matrix
 - packed storage1416
- triangular matrix factorization
 - ?pftrf439
 - ?pftri603
 - ?tftri624
- triangular system of equations1609, 1614
- tridiagonal band matrix1415
- uniform distribution1512
- unreduced symmetric tridiagonal matrix1339
 - updated upper bidiagonal matrix SVD1534
- updating sum of squares1565
 - upper Hessenberg matrix
 - computing a specified eigenvector1360
 - eigenvalues1377
 - Schur factorization1377
- utility functions and routines
 - ?labad1803
 - ?lamc11805
 - ?lamc21805
 - ?lamc31806
 - ?lamc41807
 - ?lamc51808
 - ?lamch1804
 - chla_transtype1808
 - ieeack1802
 - iladiag1809
 - ilaenv1798
 - ilaprec1810
 - ilatrans1810
 - ilauplo1811
 - ilaver1798
- iparmq1800
- xerbla_array1812
- Laplace2639
- Laplace problem
 - three-dimensional2915
 - two-dimensional2912
- largest absolute value of element
 - complex Hermitian matrix
 - packed storage1410
 - complex Hermitian matrix in RFP format1700
 - complex Hermitian tridiagonal matrix1411
 - complex symmetric matrix1412
 - general rectangular matrix1402, 2148
 - general tridiagonal matrix1403
 - Hermitian band matrix1407
 - real symmetric matrix1412, 2151
 - real symmetric matrix in RFP format1698
 - real symmetric tridiagonal matrix1411
 - symmetric band matrix1405
 - symmetric matrix
 - packed storage1408
 - trapezoidal matrix1418
 - triangular band matrix1415
 - triangular matrix
 - packed storage1416
 - upper Hessenberg matrix1404, 2149
- leading dimension3147
- leapfrog method2599
- LeapfrogStream2621
- least squares problems765
- length. dimension3143
- Library Version Obtaining2979
- library version string2979
- linear combination of distributed vectors2820
- linear combination of vectors56, 388
- Linear Congruential Generator2596
- linear equations, solving
 - tridiagonal symmetric positive-definite matrix
 - LAPACK716
 - ScaLAPACK2027
 - band matrix
 - LAPACK664, 666
 - ScaLAPACK2010
 - banded matrix
 - extra precise iterative refinement
 - LAPACK672
 - extra precise iterative refinement545, 1726, 1763
 - LAPACK672
 - Cholesky-factored matrix
 - LAPACK478
 - ScaLAPACK1848
 - diagonally dominant tridiagonal matrix
 - LAPACK471, 686
 - diagonally dominant-like matrix
 - banded1842
 - tridiagonal1845
 - general band matrix
 - ScaLAPACK2012
 - general matrix
 - band storage467, 1840
 - extra precise iterative refinement536
 - extra precise iterative refinement1740
 - general tridiagonal matrix
 - ScaLAPACK2015
 - Hermitian indefinite matrix
 - extra precise iterative refinement
 - LAPACK746
 - extra precise iterative refinement1751
 - LAPACK746
 - Hermitian matrix

- error bounds742, 761
- packed storage494, 759, 761
- Hermitian positive-definite matrix
 - band storage
 - LAPACK709
 - ScaLAPACK2025
 - error bounds
 - LAPACK691
 - ScaLAPACK2019
 - extra precise iterative refinement
 - LAPACK696
 - LAPACK
 - linear equations, solving
 - multiple right-hand sides
 - error bounds682
 - symmetric positive-definite matrix
 - LAPACK469, 480, 680, 682
 - positional auxiliary1458
 - ScaLAPACK auxiliary2329
 - tridiagonal symmetric positive-definite matrix
 - error bounds718
 - Linear Least Squares (LLS) Problems1045
- packed storage476, 703, 705
- ScaLAPACK2019
- Hermitian positive-definite tridiagonal linear equations2331
- Hermitian positive-definite tridiagonal matrix1851
- multiple right-hand sides
 - band matrix
 - LAPACK664, 666
 - ScaLAPACK2010
 - banded matrix
 - LAPACK672
 - diagonally dominant tridiagonal matrix686
 - Hermitian indefinite matrix
 - LAPACK746
 - Hermitian matrix737, 740, 759
 - Hermitian positive-definite matrix
 - band storage709
 - square matrix
 - LAPACK646, 650, 656
 - ScaLAPACK2003, 2005
 - symmetric indefinite matrix
 - LAPACK730
 - symmetric matrix721, 723, 754
 - symmetric positive-definite matrix
 - band storage709
 - symmetric/Hermitian positive-definite matrix
 - LAPACK696
 - tridiagonal matrix680, 682
 - overestimated or underestimated system2030
 - square matrix
 - error bounds
 - LAPACK650, 666
 - ScaLAPACK2005
 - extra precise iterative refinement
 - LAPACK656
 - LAPACK646, 650, 656
 - ScaLAPACK2003, 2005
 - symmetric indefinite matrix
 - extra precise iterative refinement
 - LAPACK730
 - extra precise iterative refinement1784
 - LAPACK730
 - symmetric matrix
 - error bounds726, 756
 - packed storage492, 754, 756
 - symmetric positive-definite matrix
 - band storage
 - LAPACK709
 - ScaLAPACK2025
 - error bounds
 - LAPACK691
 - ScaLAPACK2019
 - extra precise iterative refinement
 - LAPACK557, 696
 - LAPACK688, 691, 696
 - packed storage476, 703, 705
 - ScaLAPACK2017, 2019
 - symmetric positive-definite tridiagonal linear equations2331
 - triangular matrix
 - band storage501, 2250
 - packed storage499
 - tridiagonal Hermitian positive-definite matrix
 - error bounds718
 - LAPACK716
 - ScaLAPACK2027
 - tridiagonal matrix
 - error bounds682
 - LoadStreamF2617
 - LoadStreamM2620
 - Lognormal2648
 - LQ factorization
 - computing the elements of
 - orthogonal matrix Q793
 - real orthogonal matrix Q1900
 - unitary matrix Q797, 1902
 - general rectangular matrix1310, 2119
 - lsame2991
 - lsamen2991
 - LU factorization
 - band matrix
 - blocked algorithm2327
 - unblocked algorithm2326
 - diagonally dominant tridiagonal matrix434
 - diagonally dominant-like tridiagonal matrix1830
 - general band matrix1305
 - general matrix1323, 2128
 - solving linear equations
 - general matrix1320
 - square matrix2005
 - tridiagonal matrix1324, 1373
 - triangular band matrix2106
 - tridiagonal band matrix2108
 - tridiagonal matrix432, 1370, 2328
 - with complete pivoting1322, 1608
 - with partial pivoting1323, 2128

M

- machine parameters
 - LAPACK1804
 - ScaLAPACK2336
- matrices, copying
 - general matrix2338
 - trapezoidal matrix2340
- matrix arguments
 - column-major ordering3143, 3146
 - example3147
 - leading dimension3147
 - number of columns3147
 - number of rows3147
 - transposition parameter3147
- matrix block
 - QR factorization
 - with pivoting1430
- matrix converters

- mkl_?csrbsr314
- mkl_?csrcoo311
- mkl_?csrsc317
- mkl_?csrdia320
- mkl_?csrsky323
- mkl_?dnscsr308
- matrix equation
 - AX = B151, 422, 464, 1696, 1821, 1838
- matrix one-dimensional substructures3143
- matrix-matrix operation
 - product
 - general distributed matrix2868
 - general matrix126, 131, 397, 400
 - rank-2k update
 - Hermitian distributed matrix2874
 - Hermitian matrix137
 - symmetric distributed matrix2881
 - symmetric matrix145
 - rank-k update
 - Hermitian matrix135
 - symmetric distributed matrix2879
 - rank-n update
 - symmetric matrix143
 - scalar-matrix-matrix product
 - Hermitian distributed matrix2870
 - Hermitian matrix129
 - symmetric distributed matrix2877
 - symmetric matrix140
- matrix-matrix operation:scalar-matrix-matrix product
 - triangular distributed matrix2888
 - triangular matrix148
- matrix-vector operation
 - product
 - Hermitian matrix85, 88, 93
 - real symmetric matrix101, 107
 - triangular matrix112, 117, 121
 - rank-1 update
 - Hermitian matrix90, 95
 - real symmetric matrix103, 109
 - rank-2 update
 - Hermitian matrix92, 97
 - symmetric matrix105, 110
- matrix-vector operation:product
 - Hermitian matrix
 - band storage85
 - packed storage93
 - real symmetric matrix
 - packed storage101
 - symmetric matrix
 - band storage99
 - triangular matrix
 - band storage112
 - packed storage117
- matrix-vector operation:rank-1 update
 - Hermitian matrix
 - packed storage95
 - real symmetric matrix
 - packed storage103
- matrix-vector operation:rank-2 update
 - Hermitian matrix
 - packed storage97
 - symmetric matrix
 - packed storage105
- mkl_?bsrgemv178
- mkl_?bsrmm257
- mkl_?bsrmv230
- mkl_?bsrsm279
- mkl_?bsrsv243
- mkl_?bsrsymv187
- mkl_?bsrtrsv196
- mkl_?coogemv180
- mkl_?coomm265
- mkl_?coomv237
- mkl_?coosm276
- mkl_?coosv250
- mkl_?coosymv189
- mkl_?cootrsv199
- mkl_?cscmm261
- mkl_?cscmv233
- mkl_?cscsm272
- mkl_?cscsv247
- mkl_?csradd326
- mkl_?csrbsr314
- mkl_?csrcoo311
- mkl_?csrsc317
- mkl_?csrdia320
- mkl_?csrsgemv175
- mkl_?csrmm253
- mkl_?csrmultcsr330
- mkl_?csrmultd334
- mkl_?csrmmv226
- mkl_?csrsky323
- mkl_?csrsm268
- mkl_?csrsv240
- mkl_?csrsymv184
- mkl_?csrtrsv193
- mkl_?diagemv182
- mkl_?diamm295
- mkl_?diamv282
- mkl_?diasm302
- mkl_?diasv289
- mkl_?diasymv191
- mkl_?diatrsv202
- mkl_?dnscsr308
- mkl_?imatcopy404
- mkl_?omatadd412
- mkl_?omatcopy407
- mkl_?omatcopy2409
- mkl_?skymm298
- mkl_?skymv286
- mkl_?skysm305
- mkl_?skysv292
- mkl_?spffrt2463
- mkl_?spffrtx463
- mkl_alloc2999
- MKL_alloc2999
- mkl_cbwr_get3032
- mkl_cbwr_get_auto_branch3033
- mkl_cbwr_set3031
- mkl_cgetrfnpi427
- mkl_cspblas_?bsrgemv207
- mkl_cspblas_?bsrsymv214
- mkl_cspblas_?bsrtrsv221
- mkl_cspblas_?coogemv209
- mkl_cspblas_?coosymv216
- mkl_cspblas_?csrsgemv204
- mkl_cspblas_?csrsymv211
- mkl_cspblas_?csrtrsv218
- mkl_cspblas_?dcootrsv224
- mkl_ctppack1793
- mkl_ctpunpack1795
- mkl_dgetrfnpi427
- mkl_disable_fast_mm2996
- MKL_Disable_Fast_MM2996
- mkl_domain_get_max_threads2986
- MKL_Domain_Get_Max_Threads2986
- mkl_domain_set_num_threads2981
- mkl_dtpack1793
- mkl_dtpunpack1795
- mkl_enable_instructions3037

MKL_Enable_Instructions3037
 mkl_finalize3042
 mkl_free
 usage example3003
 MKL_free3001
 mkl_free_buffers2995
 MKL_Free_Buffers2995
 mkl_get_clocks_frequency2994
 MKL_Get_Clocks_Frequency2994
 mkl_get_cpu_clocks2993
 MKL_Get_Cpu_Clocks2993
 mkl_get_cpu_frequency2993
 MKL_Get_Cpu_Frequency2993
 mkl_get_dynamic2987
 MKL_Get_Dynamic2987
 mkl_get_max_cpu_frequency2994
 MKL_Get_Max_Cpu_Frequency2994
 mkl_get_max_threads2985
 MKL_Get_Max_Threads2985
 mkl_get_version_string2979
 mkl_malloc
 usage example3003
 MKL_malloc2999
 mkl_mem_stat
 usage example3003
 MKL_Mem_Stat2997
 mkl_mic_clear_status3027
 mkl_mic_disable3010
 mkl_mic_enable3010
 mkl_mic_free_memory3016
 mkl_mic_get_cpuinfo3029
 mkl_mic_get_device_count3011
 mkl_mic_get_flags3024
 mkl_mic_get_meminfo3028
 mkl_mic_get_resource_limit3022
 mkl_mic_get_status3025
 mkl_mic_get_workdivision3013
 mkl_mic_register_memory3017
 mkl_mic_set_device_num_threads3018
 mkl_mic_set_flags3024
 mkl_mic_set_max_memory3015
 mkl_mic_set_offload_report3023
 mkl_mic_set_resource_limit3019
 mkl_mic_set_workdivision3012
 mkl_pardiso_pivot2361
 mkl_peak_mem_usage2998
 mkl_progress3036
 mkl_realloc3000
 MKL_realloc3000
 mkl_set_dynamic2984
 MKL_Set_Dynamic2984
 mkl_set_env_mode3039
 mkl_set_exit_handler2990
 mkl_set_interface_layer3005
 mkl_set_memory_limit3002
 mkl_set_mpi3040
 mkl_set_num_threads2981
 MKL_Set_Num_Threads2981
 mkl_set_num_threads_local2982
 MKL_Set_Num_Threads_Local2982
 mkl_set_pardiso_pivot3009
 mkl_set_progress3008
 mkl_set_threading_layer3006
 mkl_set_xerbla3007
 mkl_sgetrfnpi427
 mkl_sparse_c_add382
 mkl_sparse_c_create_bsr345
 mkl_sparse_c_create_coo343
 mkl_sparse_c_create_csc342
 mkl_sparse_c_create_csr340
 mkl_sparse_c_export_bsr354
 mkl_sparse_c_export_csr352
 mkl_sparse_c_mm375
 mkl_sparse_c_mv370
 mkl_sparse_c_set_value357
 mkl_sparse_c_spmmd385
 mkl_sparse_c_trsm379
 mkl_sparse_c_trsv373
 mkl_sparse_convert_bsr351
 mkl_sparse_convert_csr350
 mkl_sparse_copy347
 mkl_sparse_d_add382
 mkl_sparse_d_create_bsr345
 mkl_sparse_d_create_coo343
 mkl_sparse_d_create_csc342
 mkl_sparse_d_create_csr340
 mkl_sparse_d_export_bsr354
 mkl_sparse_d_export_csr352
 mkl_sparse_d_mm375
 mkl_sparse_d_mv370
 mkl_sparse_d_set_value357
 mkl_sparse_d_spmmd385
 mkl_sparse_d_trsm379
 mkl_sparse_d_trsv373
 mkl_sparse_destroy349
 mkl_sparse_optimize369
 mkl_sparse_s_add382
 mkl_sparse_s_create_bsr345
 mkl_sparse_s_create_coo343
 mkl_sparse_s_create_csc342
 mkl_sparse_s_create_csr340
 mkl_sparse_s_export_bsr354
 mkl_sparse_s_export_csr352
 mkl_sparse_s_mm375
 mkl_sparse_s_mv370
 mkl_sparse_s_set_value357
 mkl_sparse_s_spmmd385
 mkl_sparse_s_trsm379
 mkl_sparse_s_trsv373
 mkl_sparse_set_memory_hint367
 mkl_sparse_set_mm_hint363
 mkl_sparse_set_mv_hint358
 mkl_sparse_set_sm_hint365
 mkl_sparse_set_sv_hint360
 mkl_sparse_spm384
 mkl_sparse_z_add382
 mkl_sparse_z_create_bsr345
 mkl_sparse_z_create_coo343
 mkl_sparse_z_create_csc342
 mkl_sparse_z_create_csr340
 mkl_sparse_z_export_bsr354
 mkl_sparse_z_export_csr352
 mkl_sparse_z_mm375
 mkl_sparse_z_mv370
 mkl_sparse_z_set_value357
 mkl_sparse_z_spmmd385
 mkl_sparse_z_trsm379
 mkl_sparse_z_trsv373
 mkl_stppack1793
 mkl_stpunpack1795
 mkl_thread_free_buffers2996
 MKL_Thread_Free_Buffers2996
 mkl_verbose3039
 mkl_zgetrfnpi427
 mkl_ztppack1793
 mkl_ztpunpack1795
 MPI1819
 Multiplicative_Congruential_Generator2596

N

naming conventions
 BLAS51
 LAPACK1821
 Nonlinear Optimization Solvers2952
 PBLAS2816
 Sparse BLAS Level 1154
 Sparse BLAS Level 2165
 Sparse BLAS Level 3165
 VM2484
 negative eigenvalues2145
 NegBinomial2673
 NewStream2607
 NewStreamEx2608
 NewTaskX1D2688
 Nonsymmetric Eigenproblems1140

O

off-diagonal elements
 initialization2208
 LAPACK1552
 ScaLAPACK2208
 orthogonal matrix
 CS decomposition
 LAPACK1034, 1039, 1184, 1188, 1622,
 1625, 1628, 1631, 1635, 1637
 from LQ factorization
 LAPACK1642
 ScaLAPACK2231
 from QL factorization
 LAPACK1640, 1645
 ScaLAPACK2227, 2236
 from QR factorization
 LAPACK1641
 ScaLAPACK2229
 from RQ factorization
 LAPACK1644
 ScaLAPACK2234
 multiplication2095

P

p?agemv2834
 p?ahemv2844
 p?amax2818
 p?asum2819
 p?asymv2851
 p?atrmv2859
 p?axpy2820
 p?copy2822
 p?dbsv2012
 p?dbtrf1827
 p?dbtrs1842
 p?dbtrsv2106
 p?dot2823
 p?dotc2825
 p?dotu2826
 p?dtsv2015
 p?dttrf1830
 p?dttrs1845
 p?dttrsv2108
 p?gbsv2010
 p?gbtrf1825
 p?gbtrs1840
 p?geadd2864
 p?gebd22114
 p?gebrd1986
 p?gecon1855
 p?geequ1878
 p?gehd22117
 p?gehrd1972
 p?gelq22119
 p?gelqf1897
 p?gels2030
 p?gemm2868
 p?gemr2d2338
 p?gemv2831
 p?geql22122
 p?geqlf1909
 p?geqpf1885
 p?geqr22124
 p?geqrf1883
 p?ger2836
 p?gerc2838
 p?gerfs1863
 p?gerq22126
 p?gerqf1921
 p?geru2840
 p?gesv2003
 p?gesvd2068
 p?gesvx2005
 p?getf22128
 p?getrf1823
 p?getri1873
 p?getrs1838
 p?ggqrf1942
 p?ggrqf1946
 p?heev2050
 p?heevd2053
 p?heevx2061
 p?hegst2001
 p?hegvx2079
 p?hemm2870
 p?hemv2842
 p?her2846
 p?her22847
 p?her2k2874
 p?herk2872
 p?hetrd1957
 p?labad2334
 p?labrd2130
 p?lachkieee2335
 p?lacon2134
 p?laconsb2135
 p?lacp22136
 p?lacp32138
 p?lacpy2139
 p?laevswp2141
 p?lahqr1981
 p?lahrd2142
 p?laiect2145
 p?lamch2336
 p?lange2148
 p?lanhs2149
 p?lantr2153
 p?lapiv2155
 p?laqge2157
 p?laqsy2176
 p?lared1d2178
 p?lared2d2179
 p?larf2180
 p?larfb2183
 p?larfc2187
 p?larfg2190
 p?larft2191
 p?larz2194
 p?larzb2197

p?larzt2203
 p?lascl2206
 p?laset2208
 p?lasmsub2209
 p?lasnbt2337
 p?lassq2211
 p?laswp2212
 p?latra2214
 p?latrd2215
 p?latrz2221
 p?lauu2223
 p?lauum2225
 p?lawil2226
 p?max12101
 p?nrm22827
 p?org2l/p?ung2l2227
 p?org2r/p?ung2r2229
 p?orgl2/p?ungl22231
 p?orglq1900
 p?orgql1911
 p?orgqr1888
 p?orgr2/p?ungr22234
 p?orgrq1923
 p?orm2l/p?unm2l2236
 p?orm2r/p?unm2r2239
 p?ormbr1990
 p?ormhr1975
 p?orml2/p?unml22243
 p?ormlq1903
 p?ormql1915
 p?ormqr1892
 p?ormr2/p?unmr22246
 p?ormrq1927
 p?ormrz1936
 p?ormtr1954
 p?pbsv2025
 p?pbtrf1833
 p?pbtrs1848
 p?pbtrsv2250
 p?pocon1857
 p?poequ1880
 p?porfs1866
 p?posv2017
 p?posvx2019
 p?potf22257
 p?potrf1832
 p?potri1875
 p?potrs1847
 p?ptsv2027
 p?pttrf1836
 p?pttrs1851
 p?pttrsv2254
 p?rscl2261
 p?scal2828
 p?stebz1964
 p?stein1968
 p?sum12105
 p?swap2829
 p?syev2034
 p?syevd2036
 p?syevx2043
 p?sygs2/p?hegs22262
 p?sygst1999
 p?sygvx2072
 p?symm2877
 p?symv2849
 p?syr2853
 p?syr22855
 p?syr2k2881
 p?syrk2879
 p?sytd2/p?hetd22264
 p?sytrd1951
 p?tradd2866
 p?tran2884
 p?tranc2886
 p?tranu2885
 p?trcon1860
 p?trmm2888
 p?trmr2d2340
 p?trmv2857
 p?trrfs1870
 p?trsm2890
 p?trsv2861
 p?trti22276
 p?trtri1877
 p?trtrs1853
 p?tzrzf1933
 p?unglq1902
 p?ungql1913
 p?ungqr1890
 p?ungrq1925
 p?unmbr1994
 p?unmhr1978
 p?unmlq1906
 p?unmql1918
 p?unmqr1895
 p?unmrq1930
 p?unmrz1939
 p?unmtr1961
 Packed formats2772
 packed storage scheme3145
 parallel direct solver (PARDISO)2345
 Parallel Direct Sparse Solver for Clusters2384
 parallel direct sparse solver interface
 mkl_pardiso_pivot2361
 pardiso2350
 pardiso_642358
 pardiso_getdiag2362
 pardiso_getenv2359
 pardiso_handle_delete2365
 pardiso_handle_delete_642367
 pardiso_handle_restore2364
 pardiso_handle_restore_642366
 pardiso_handle_store2363
 pardiso_handle_store_642366
 pardiso_setenv2359
 pardisoinit2357
 parallel direct sparse solver interface for clusters
 cluster_sparse_solver2385
 parameters
 for a Givens rotation65
 modified Givens transformation68
 pardiso2350
 pardiso iparm parameter2372
 PARDISO parameters2368
 pardiso_642358
 PARDISO_DATA_TYPE2383
 pardiso_getdiag2362
 pardiso_getenv2359
 pardiso_handle_delete2365
 pardiso_handle_delete_642367
 pardiso_handle_restore2364
 pardiso_handle_restore_642366
 pardiso_handle_store2363
 pardiso_handle_store_642366
 pardiso_setenv2359
 PARDISO* solver2345
 pardisoinit2357
 Partial Differential Equations support
 Helmholtz problem on a sphere2912

Poisson problem on a sphere2912
 three-dimensional Helmholtz problem2914
 three-dimensional Laplace problem2915
 three-dimensional Poisson problem2915
 two-dimensional Helmholtz problem2911
 two-dimensional Laplace problem2912
 two-dimensional Poisson problem2912
 PBLAS Level 1 functions
 p?amax2818
 p?asum2819
 p?dot2823
 p?dotc2825
 p?dotu2826
 p?nrm22827
 PBLAS Level 1 routines
 p?amax2818
 p?asum2818
 p?axpy2818, 2820
 p?copy2818, 2822
 p?dot2818
 p?dotc2818
 p?dotu2818
 p?nrm22818
 p?scal2818, 2828
 p?swap2818, 2829
 PBLAS Level 2 routines
 ?agemv2830
 ?asymv2830
 ?gemv2830
 ?ger2830
 ?gerc2830
 ?geru2830
 ?hemv2830
 ?her2830
 ?her22830
 ?symv2830
 ?syr2830
 ?syr22830
 ?trmv2830
 ?trsv2830
 p?agemv2834
 p?ahemv2844
 p?asymv2851
 p?atrmv2859
 p?gemv2831
 p?ger2836
 p?gerc2838
 p?geru2840
 p?hemv2842
 p?her2846
 p?her22847
 p?symv2849
 p?syr2853
 p?syr22855
 p?trmv2857
 p?trsv2861
 PBLAS Level 3 routines
 p?geadd2864
 p?gemm2863, 2868
 p?hemm2863, 2870
 p?her2k2863, 2874
 p?herk2863, 2872
 p?symm2863, 2877
 p?syr2k2863, 2881
 p?syrk2863, 2879
 p?tradd2866
 p?tran2884
 p?tranc2886
 p?tranu2885
 p?trmm2863, 2888
 p?trsm2863, 2890
 PBLAS routines
 routine groups2815
 pcagemv2834
 pcahemv2844
 pcamax2818
 pcatrmv2859
 pcaxpy2820
 pccopy2822
 pcdotc2825
 pcdotu2826
 pcgeadd2864
 pcgecon1855
 pcgemm2868
 pcgemv2831
 pcgerc2838
 pcgeru2840
 pcheevr2055
 pchemm2870
 pchemv2842
 pcher2846
 pcher22847
 pcher2k2874
 pcherk2872
 pcnrm22827
 pcscal2828
 pcscal2828
 pcswap2829
 pcsymm2877
 pcsyr2k2881
 pcsyrk2879
 pctradd2866
 pctranu2885
 pctrmm2888
 pctrmv2857
 pctrsm2890
 pctrsv2861
 pdagemv2834
 pdamax2818
 pdasum2819
 pdasymv2851
 pdatrmv2859
 pdaxpy2820
 pdcopy2822
 pddot2823
 PDE support2893
 pdgeadd2864
 pdgebal2112
 pdgecon1855
 pdgemm2868
 pdgemv2831
 pdger2836
 pdhseqr1983
 pdlaiectb2145
 pdlaiectl2145
 pdlamve2146
 pdlaqr02160
 pdlaqr12162
 pdlaqr22165
 pdlaqr32168
 pdlaqr42171
 pdlaqr52174
 pdnrm22827
 pdrot2259
 pdscal2828
 pdswap2829
 pdsyevr2039
 pdsymm2877
 pdsymv2849
 pdsyr2853

pdsyr22855
 pdsyr2k2881
 pdsyrk2879
 pdtradd2866
 pdtran2884
 pdtranc2886
 pdtrmm2888
 pdtrmv2857
 pdtrord2268
 pdtrsen2271
 pdtrsm2890
 pdtrsv2861
 pdzasum2819
 permutation matrix3126
 picopy2822
 pivoting matrix rows or columns2155
 planar rotation2259
 pmpcol2102
 pmpim22103
 points rotation
 in the modified plane66
 in the plane64
 Poisson2669
 Poisson problem
 on a sphere2912
 three-dimensional2915
 two-dimensional2912
 Poisson Solver
 routines
 ?_commit_Helmholtz_2D2920
 ?_commit_Helmholtz_3D2920
 ?_commit_sph_np2929
 ?_commit_sph_p2929
 ?_Helmholtz_2D2923
 ?_Helmholtz_3D2923
 ?_init_Helmholtz_2D2918
 ?_init_Helmholtz_3D2918
 ?_init_sph_np2927
 ?_init_sph_p2927
 ?_sph_np2931
 ?_sph_p2931
 free_Helmholtz_2D2926
 free_Helmholtz_3D2926
 free_sph_np2933
 free_sph_p2933
 structure2910
 Poisson Solver Interface2909
 PoissonV2671
 pprfs563
 pptrs476
 preconditioned Jacobi SVD1167
 preconditioners based on incomplete LU factorization
 dcsrilu02443
 dcsrilit2445
 Preconditioners Interface Description2442
 process grid1819, 2815
 product
 distributed matrix-vector
 general matrix2831, 2834
 distributed vector-scalar2828
 matrix-vector
 distributed Hermitian matrix2842, 2844
 distributed symmetric matrix2849, 2851
 distributed triangular matrix2857, 2859
 general matrix75, 78, 389, 391, 1734
 Hermitian indefinite matrix1746
 Hermitian matrix85, 88, 93
 real symmetric matrix101, 107
 symmetric indefinite matrix1778
 triangular matrix112, 117, 121
 scalar-matrix
 general distributed matrix2868
 general matrix126, 131, 397, 400
 Hermitian distributed matrix2870
 Hermitian matrix129
 scalar-matrix-matrix
 general distributed matrix2868
 general matrix126, 131, 397, 400
 Hermitian distributed matrix2870
 Hermitian matrix129
 symmetric distributed matrix2877
 symmetric matrix140
 triangular distributed matrix2888
 triangular matrix148
 vector-scalar69
 product:matrix-vector
 general matrix
 band storage75
 Hermitian matrix
 band storage85
 packed storage93
 real symmetric matrix
 packed storage101
 symmetric matrix
 band storage99
 triangular matrix
 band storage112
 packed storage117
 psagemv2834
 psamax2818
 psasum2819
 psasymv2851
 psatrmv2859
 psaxpy2820
 pscasum2819
 pscopy2822
 psdot2823
 pseudorandom numbers2593
 psgeadd2864
 psgebal2112
 psgecon1855
 psgemm2868
 psgemv2831
 psger2836
 pshseqr1983
 pslaiect2145
 psclamve2146
 pslaqr02160
 pslaqr12162
 pslaqr22165
 pslaqr32168
 pslaqr42171
 pslaqr52174
 psnrm22827
 psrot2259
 psscal2828
 psswap2829
 pssyevr2039
 pssymm2877
 pssymv2849
 pssyr2853
 pssyr22855
 pssyr2k2881
 pssyrk2879
 pstradd2866
 pstran2884
 pstranc2886
 pstrmm2888
 pstrmv2857
 pstord2268

pstrsen2271
 pstrsm2890
 pstrsv2861
 pxfcrbla2989
 pzagemv2834
 pzahemv2844
 pzamax2818
 pzatrmv2859
 pzaxpy2820
 pzcopy2822
 pzdotc2825
 pzdotu2826
 pzdsal2828
 pzgeadd2864
 pzgecon1855
 pzgemm2868
 pzgemv2831
 pzgerc2838
 pzgeru2840
 pzheevr2055
 pzhemm2870
 pzhemv2842
 pzher2846
 pzher22847
 pzher2k2874
 pzherk2872
 pznrm22827
 pzscal2828
 pzswap2829
 pzsymm2877
 pzsyk2k2881
 pzsyk2879
 pztradd2866
 pztranu2885
 pztrmm2888
 pztrmv2857
 pztrsm2890
 pztrsv2861

Q

QL factorization

- computing the elements of
 - complex matrix Q805
 - orthogonal matrix Q1911
 - real matrix Q803
 - unitary matrix Q1913
- general rectangular matrix
 - LAPACK1311
 - ScaLAPACK2122
- multiplying general matrix by
 - orthogonal matrix Q1915
 - unitary matrix Q1918

QR factorization

- applying matrix obtained from block reflector to
 - general matrix
 - orthogonal/unitary matrix Q839
- computing the elements of
 - orthogonal matrix Q781, 1888
 - unitary matrix Q785, 1890
- general rectangular matrix
 - LAPACK1313, 1314, 1319
 - ScaLAPACK2124, 2126
- multiplying general matrix by
 - orthogonal/unitary matrix Q774
- with pivoting
 - ScaLAPACK1885

quasi-random numbers2593

quasi-triangular matrix

LAPACK943, 989
 ScaLAPACK1971
 quasi-triangular system of equations1456

R

random number generators2593
 random stream2602
 random stream descriptor2595
 Random Streams2602
 rank-1 update

- conjugated, distributed general matrix2838
- conjugated, general matrix82
- distributed general matrix2836
- distributed Hermitian matrix2846
- distributed symmetric matrix2853
- general matrix81
 - Hermitian matrix
 - packed storage95
 - real symmetric matrix
 - packed storage103
- unconjugated, distributed general matrix2840
- unconjugated, general matrix84

rank-2 update

- distributed Hermitian matrix2847
- distributed symmetric matrix2855
 - Hermitian matrix
 - packed storage97
 - symmetric matrix
 - packed storage105

rank-2k update

- Hermitian distributed matrix2874
- Hermitian matrix137
- symmetric distributed matrix2881
- symmetric matrix145

rank-k update

- distributed Hermitian matrix2872
- Hermitian matrix135
- symmetric distributed matrix2879

rank-n update

- symmetric matrix143

Rayleigh2645

RCI CG Interface2413

RCI CG sparse solver routines

- dcg2428, 2432
- dcg_check2427
- dcg_get2429
- dcg_init2426
- dcgmrhs_check2431
- dcgmrhs_get2434
- dcgmrhs_init2430

RCI FGMRES Interface2418

RCI FGMRES sparse solver routines

- dfgmres_check2436
- dfgmres_get2439
- dfgmres_init2435

RCI GFMRES sparse solver routines

- dfgres2437

RCI ISS2411

RCI ISS interface2411

RCI ISS sparse solver routines

- implementation details2440

real matrix

- QR factorization
 - with pivoting1432

real symmetric matrix

- 1-norm value1412
- Frobenius norm1412
- infinity- norm1412

largest absolute value of element1412
 real symmetric tridiagonal matrix
 1-norm value1411
 Frobenius norm1411
 infinity- norm1411
 largest absolute value of element1411
 reducing generalized eigenvalue problems
 LAPACK930
 ScaLAPACK1999
 reduction to upper Hessenberg form
 general matrix2117
 general square matrix1308
 refining solutions of linear equations
 band matrix543
 banded matrix545, 1726, 1763
 general matrix534, 536, 1740, 1863
 Hermitian indefinite matrix581, 1751
 Hermitian matrix
 packed storage589
 Hermitian positive-definite matrix
 band storage565
 packed storage563
 symmetric indefinite matrix573, 1784
 symmetric matrix
 packed storage587
 symmetric positive-definite matrix
 band storage565
 packed storage563
 symmetric/Hermitian positive-definite distributed
 matrix1866
 tridiagonal matrix552
 Relatively robust representation (RRR)2302
 reordering of matrices3127
 Reverse Communication Interface2411
 rotation
 of points in the modified plane66
 of points in the plane64
 of sparse vectors162
 parameters for a Givens rotation65
 parameters of modified Givens transformation68
 rotation, planar2259
 routine name conventions
 BLAS51
 Nonlinear Optimization Solvers2952
 PBLAS2816
 Sparse BLAS Level 1154
 Sparse BLAS Level 2165
 Sparse BLAS Level 3165
 RQ factorization
 computing the elements of
 complex matrix Q815
 orthogonal matrix Q1923
 real matrix Q814
 unitary matrix Q1925

S

SaveStreamF2616
 SaveStreamM2619
 sbbcscd1034
 sbdsdc864
 ScaLAPACK1819
 ScaLAPACK routines
 1D array redistribution2178, 2179
 auxiliary routines
 ?combamax12104
 ?dbtf22326
 ?dbtrf2327
 ?dttrf2328
 ?dttrsv2329
 ?lamsh2278
 ?laref2285
 ?lasorte2310
 ?lasrt22311
 ?pttrsv2331
 ?stein22324
 ?steqr22332
 bdlaapp2095
 bdlaexc2096
 bdtrexc2098
 bslaapp2095
 bslaexc2096
 bstrexc2098
 dlaqr62279
 dlar1va2282
 dlarrb22287
 dlarrd22290
 dlarre22293
 dlarre2a2298
 dlarrf22302
 dlarrv22304
 dstegr22312
 dstegr2a2315
 dstegr2b2319
 p?dbtrsv2106
 p?dttrsv2108
 p?gebd22114
 p?gehd22117
 p?gelq22119
 p?geql22122
 p?geqr22124
 p?gerq22126
 p?getf22128
 p?labrd2130
 p?lacgv2100
 p?lacon2134
 p?laconsb2135
 p?lacz22136
 p?lacz32138
 p?lacpy2139
 p?laevswp2141
 p?lahrd2142
 p?laiect2145
 p?lange2148
 p?lanhs2149
 p?lansy, p?lanhe2151
 p?lantr2153
 p?lapiv2155
 p?laqge2157
 p?laqsy2176
 p?lared1d2178
 p?lared2d2179
 p?larf2180
 p?larfb2183
 p?larfc2187
 p?larfg2190
 p?larft2191
 p?larz2194
 p?larzb2197
 p?larzc2201
 p?larzt2203
 p?lascl2206
 p?laset2208
 p?lasmsub2209
 p?lassq2211
 p?laswp2212
 p?latra2214
 p?latrd2215
 p?latrs2219

- p?latrz2221
- p?lauu22223
- p?lauum2225
- p?lawil2226
- p?max12101
- p?org2l/p?ung2l2227
- p?org2r/p?ung2r2229
- p?orgl2/p?ungl22231
- p?orgr2/p?ungr22234
- p?orm2l/p?unm2l22236
- p?orm2r/p?unm2r22239
- p?orml2/p?unml22243
- p?ormr2/p?unmr22246
- p?pbtrsv22250
- p?potf22257
- p?pttrsv22254
- p?rscl22261
- p?sum12105
- p?sygs2/p?hegs22262
- p?sytd2/p?hetd22264
- p?trti22276
- pdgebal2112
- pdhseqr1983
- pdlaiectb2145
- pdlaiectl2145
- pdlamve2146
- pdlaqr02160
- pdlaqr12162
- pdlaqr22165
- pdlaqr32168
- pdlaqr42171
- pdlaqr52174
- pdrot2259
- pdtrord2268
- pdtrsen2271
- pmpcol2102
- pmpim22103
- psgebal2112
- pshseqr1983
- pslaiect2145
- pslamve2146
- pslaqr02160
- pslaqr12162
- pslaqr22165
- pslaqr32168
- pslaqr42171
- pslaqr52174
- psrot2259
- pstrord2268
- pstrsen2271
- slaqr62279
- slar1va2282
- slarrb22287
- slarrd22290
- slarre22293
- slarre2a2298
- slarrf22302
- slarrv22304
- sstegr22312
- sstegr2a2315
- sstegr2b2319
- block reflector
 - triangular factor2191, 2203
- Cholesky factorization1836
- complex matrix
 - complex elementary reflector2201
- complex vector
 - 1-norm using true absolute value2105
- complex vector conjugation2100
- condition number estimation
 - p?gecon1855
 - p?pocon1857
 - p?trcon1860
- copying matrices
 - p?gemr2d2338
 - p?trmr2d2340
- driver routines
 - p?dbsv2012
 - p?dtsv2015
 - p?gbsv2010
 - p?gels2030
 - p?gesv2003
 - p?gesvd2068
 - p?gesvx2005
 - p?heev2050
 - p?heevd2053
 - p?heevx2061
 - p?hegvx2079
 - p?pbsv2025
 - p?posv2017
 - p?posvx2019
 - p?ptsv2027
 - p?syev2034
 - p?syevd2036
 - p?syevx2043
 - p?sygvx2072
 - pcheevr2055
 - pdsyevr2039
 - pssyevr2039
 - pzheevr2055
- error estimation
 - p?trrfs1870
- error handling
 - pxerbla2989
- general matrix
 - block reflector2197
 - elementary reflector2194
 - LU factorization2128
 - reduction to upper Hessenberg form2117
- general rectangular matrix
 - elementary reflector2180
 - LQ factorization2119
 - QL factorization2122
 - QR factorization2124
 - reduction to bidiagonal form2130
 - reduction to real bidiagonal form2114
 - row interchanges2212
 - RQ factorization2126
- generalized eigenvalue problems
 - p?hegst2001
 - p?sygst1999
- Householder matrix
 - elementary reflector2190
- LQ factorization
 - p?gelq22119
 - p?gelqf1897
 - p?orglq1900
 - p?ormlq1903
 - p?unglq1902
 - p?unmlq1906
- LU factorization
 - p?dbtrsv2106
 - p?dttrf1830
 - p?dttrsv2108
 - p?getf22128
- matrix equilibration
 - p?geequ1878

p?poequ1880
 matrix inversion
 p?getri1873
 p?potri1875
 p?trtri1877
 nonsymmetric eigenvalue problems
 p?gehrd1972
 p?lahqr1981
 p?ormhr1975
 p?unmhr1978
 QL factorization
 ?geqlf1909
 ?ungql1913
 p?geql22122
 p?orgql1911
 p?ormql1915
 p?unmql1918
 QR factorization
 p?geqpf1885
 p?geqr22124
 p?ggqrf1942
 p?orgqr1888
 p?ormqr1892
 p?ungqr1890
 p?unmqr1895
 RQ factorization
 p?gerq22126
 p?gerqf1921
 p?ggrqf1946
 p?orgrq1923
 p?ormrq1927
 p?ungrq1925
 p?unmrq1930
 RZ factorization
 p?ormrz1936
 p?tzzf1933
 p?unmrz1939
 singular value decomposition
 p?gebrd1986
 p?ormbr1990
 p?unmbr1994
 solution refinement and error
 estimation
 p?gerfs1863
 p?porfs1866
 solving linear equations
 ?dttrsv2329
 ?pttrsv2331
 p?dbtrs1842
 p?dttrs1845
 p?gbtrs1840
 p?getrs1838
 p?potrs1847
 p?pttrs1851
 p?trtrs1853
 symmetric eigenproblems
 p?hetrd1957
 p?ormtr1954
 p?stebz1964
 p?stein1968
 p?sytrd1951
 p?unmtr1961
 symmetric eigenvalue problems
 ?stein22324
 ?steqr22332
 trapezoidal matrix2221
 triangular factorization
 ?dbtrf2327
 ?dttrf2328
 p?dbtrsv2106
 p?dttrsv2108
 p?gbtrf1825
 p?getrf1823
 p?pbtrf1833
 p?potrf1832
 p?pttrf1836
 triangular system of equations2219
 updating sum of squares2211
 utility functions and routines
 p?labad2334
 p?lachkieee2335
 p?lamch2336
 p?lasnbt2337
 p?xerbla2989
 scalar-matrix product126, 129, 131, 140, 397, 400, 2868,
 2870, 2877
 scalar-matrix-matrix product
 general distributed matrix2868
 general matrix126, 131, 397, 400
 symmetric distributed matrix2877
 symmetric matrix140
 triangular distributed matrix2888
 triangular matrix148
 scaling
 general rectangular matrix2157
 symmetric/Hermitian matrix2176
 scaling factors
 general rectangular distributed matrix1878
 Hermitian positive definite distributed matrix1880
 symmetric positive definite distributed matrix1880
 scattering compressed sparse vector's elements into full
 storage form164
 Schur decomposition1006, 1009, 2160, 2171
 Schur factorization
 reordering2271
 scsum11304
 second/dsecnd2992
 Service Functions2485
 Service Routines2606
 SetInternalDecimation2694
 sgbcon506
 sgbrfsx545
 sgbsvx666
 sgbtrs467
 sgecon504
 sgejsv1167
 sgemmt394
 sgeqpf776
 sgesvj1174
 sgtrfs552
 shgeqz996
 shseqr962
 simple driver1821
 Single Dynamic Library
 mkl_set_exit_handler2990
 mkl_set_interface_layer3005
 mkl_set_pardiso_pivot3009
 mkl_set_progress3008
 mkl_set_threading_layer3006
 mkl_set_xerbla3007
 single node matrix2278
 singular value decomposition
 LAPACK842
 LAPACK routines, singular value decomposition1986
 ScaLAPACK1986, 2068
 See also LAPACK routines, singular value
 decomposition842
 Singular Value Decomposition1159
 sjacobi2971
 sjacobi_delete2970

- naming conventions165
- sparse BLAS Level 3 routines
 - mkl_?bsrsm257
 - mkl_?bsrsm279
 - mkl_?coomm265
 - mkl_?coosm276
 - mkl_?cscmm261
 - mkl_?cscsm272
 - mkl_?csradd326
 - mkl_?csrmm253
 - mkl_?csrmultcsr330
 - mkl_?csrmultd334
 - mkl_?csrsm268
 - mkl_?diamm295
 - mkl_?diasm302
 - mkl_?skymm298
 - mkl_?skysm305
- sparse BLAS routines
 - mkl_?csrbsr314
 - mkl_?csrcoo311
 - mkl_?csrsc317
 - mkl_?csrdia320
 - mkl_?csrsky323
 - mkl_?dnscsr308
- sparse matrices165
- sparse matrix165
- Sparse Matrix Checker Routines2449
- Sparse Matrix Storage Formats166
- sparse solver
 - parallel direct sparse solver interface
 - mkl_pardiso_pivot2361
 - pardiso2350
 - pardiso_642358
 - pardiso_getdiag2362
 - pardiso_getenv2359
 - pardiso_handle_delete2365
 - pardiso_handle_delete_642367
 - pardiso_handle_restore2364
 - pardiso_handle_restore_642366
 - pardiso_handle_store2363
 - pardiso_handle_store_642366
 - pardiso_setenv2359
 - pardisoinit2357
 - parallel direct sparse solver interface for clusters
 - cluster_sparse_solver2385
- Sparse Solver
 - direct sparse solver interface
 - dss_create2399
 - dss_define_structure
 - dss_define_structure2400
 - dss_delete2407
 - dss_factor2403
 - dss_factor_complex2403
 - dss_factor_real2403
 - dss_reorder2402
 - dss_solve2405
 - dss_solve_complex2405
 - dss_solve_real2405
 - dss_statistics2408
 - mkl_cvt_to_null_terminated_str2411
 - iterative sparse solver interface
 - dcg2428
 - dcg_check2427
 - dcg_get2429
 - dcg_init2426
 - dcgmrhs2432
 - dcgmrhs_check2431
 - dcgmrhs_get2434
 - dcgmrhs_init2430
 - dfgmres2437
 - dfgmres_check2436
 - dfgmres_get2439
 - dfgmres_init2435
 - preconditioners based on incomplete LU factorization
 - dcsrilu02443
 - dcsrilut2445
 - Sparse Solvers2345, 2368, 2372, 2383, 2391, 2453, 2454, 2456, 2460, 2466
 - sparse vectors
 - adding and scaling155
 - complex dot product, conjugated158
 - complex dot product, unconjugated159
 - compressed form154
 - converting to compressed form160, 161
 - converting to full-storage form164
 - full-storage form154
 - Givens rotation162
 - norm154
 - passed to BLAS level 1 routines154
 - real dot product157
 - scaling154
 - sparse_matrix_checker2449
 - sparse_matrix_init2451
 - spbt21655
 - specific hardware support
 - mkl_enable_instructions3037
 - mkl_finalize3042
 - mkl_set_mpi3040
 - Spline Methods3106
 - split Cholesky factorization (band matrices)941
 - sporfsx557
 - spotf21657
 - spprfs563
 - spptvs476
 - sptts21658
 - square matrix
 - 1-norm estimation
 - LAPACK1329, 1330
 - ScaLAPACK2134
 - srslc1659
 - sstegr22312
 - sstegr2a2315
 - sstegr2b2319
 - ssyconv1299
 - ssygs21665
 - ssyswapr1660
 - ssyswapr11663
 - ssytd21666
 - ssytf21668
 - ssytf2_rook1670
 - ssytri2612
 - ssytri2x616
 - ssytrs2489
 - stgex21674
 - stgsy21677
 - stream2602
 - strex980
 - stride_increment3143
 - strnlsp_check2955
 - strnlsp_delete2959
 - strnlsp_get2958
 - strnlsp_init2953
 - strnlsp_solve2956
 - strnlspbc_check2962
 - strnlspbc_delete2967
 - strnlspbc_get2966
 - strnlspbc_init2960
 - strnlspbc_solve2965
 - strti21680

- sum
 - of distributed vectors2820
 - of magnitudes of elements of a distributed vector2819
 - of magnitudes of the vector elements54
 - of sparse vector and full-storage vector155
 - of vectors56, 388
- sum of squares
 - updating
 - LAPACK1565
 - ScaLAPACK2211
- summary statistics
 - vsldsscompute2750
 - vsldSSCompute2750
 - vsldsseditcorparameterization2746
 - vsldSSEditCorParameterization2746
 - vsldsseditcovcor2732
 - vsldSSEditCovCor2732
 - vsldsseditcp2734
 - vsldSSEditCP2734
 - vsldsseditmissingvalues2743
 - vsldSSEditMissingValues2743
 - vsldsseditmoments2730
 - vsldSSEditMoments2730
 - vsldsseditoutliersdetection2741
 - vsldSSEditOutliersDetection2741
 - vsldsseditpartialcovcor2735
 - vsldsseditpooledcovariance2738
 - vsldSSEditPooledCovariance2738
 - vsldsseditquantiles2736
 - vsldSSEditQuantiles2736
 - vsldsseditrobustcovariance2740
 - vsldSSEditRobustCovariance2740
 - vsldsseditstreamquantiles2737
 - vsldSSEditStreamQuantiles2737
 - vsldsseditsums2731
 - vsldSSEditSums2731
 - vsldssedittask2721
 - vsldSSEditTask2721
 - vsldssnewtask2718
 - vsldSSNewTask2718
 - vsliSSedittask2721
 - vsliSSEditTask2721
 - vslsdeleteetask2751
 - vsISSDeleteTask2751
 - vslsSScompute2750
 - vsLSSSCompute2750
 - vslsSSeditcorparameterization2746
 - vsLSSEditCorParameterization2746
 - vslsSSeditcovcor2732
 - vsLSSEditCovCor2732
 - vslsSSeditcp2734
 - vsLSSEditCP2734
 - vslsSSeditmissingvalues2743
 - vsLSSEditMissingValues2743
 - vslsSSeditmoments2730
 - vsLSSEditMoments2730
 - vslsSSeditoutliersdetection2741
 - vsLSSEditOutliersDetection2741
 - vslsSSeditpartialcovcor2735
 - vslsSSeditpooledcovariance2738
 - vsLSSEditPooledCovariance2738
 - vslsSSeditquantiles2736
 - vsLSSEditQuantiles2736
 - vslsSSeditrobustcovariance2740
 - vsLSSEditRobustCovariance2740
 - vslsSSeditstreamquantiles2737
 - vsLSSEditStreamQuantiles2737
 - vslsSSeditsums2731
 - vsLSSEditSums2731
 - vslsSSedittask2721
- vslsSSEditTask2721
- vslsSSnewtask2718
- vsLSSSNewTask2718
- Summary Statistics2712
- summary statistics usage examples2752
- support functions
 - mkl_malloc2999
 - mkl_enable_instructions3037
 - mkl_finalize3042
 - mkl_free3001
 - mkl_malloc2999
 - mkl_mem_stat2997
 - mkl_mic_clear_status3027
 - mkl_mic_disable3010
 - mkl_mic_enable3010
 - mkl_mic_free_memory3016
 - mkl_mic_get_cpuinfo3029
 - mkl_mic_get_device_count3011
 - mkl_mic_get_flags3024
 - mkl_mic_get_meminfo3028
 - mkl_mic_get_resource_limit3022
 - mkl_mic_get_status3025
 - mkl_mic_get_workdivision3013
 - mkl_mic_register_memory3017
 - mkl_mic_set_device_num_threads3018
 - mkl_mic_set_flags3024
 - mkl_mic_set_max_memory3015
 - mkl_mic_set_offload_report3023
 - mkl_mic_set_resource_limit3019
 - mkl_mic_set_workdivision3012
 - mkl_peak_mem_usage2998
 - mkl_progress3036
 - mkl_realloc3000
 - mkl_set_env_mode3039
 - mkl_set_mpi3040
 - mkl_verbose3039
- Support Functions
 - exception handling2988
 - handling fatal errors2990
- support functions for CNR
 - mkl_cbwr_get3032
 - mkl_cbwr_get_auto_branch3033
 - mkl_cbwr_set3031
- support routines
 - mkl_disable_fast_mm2996
 - mkl_free_buffers2995
 - mkl_set_memory_limit3002
 - mkl_thread_free_buffers2996
 - progress information3036
- SVD (singular value decomposition)
 - LAPACK842
 - ScaLAPACK1986
- swapping adjacent diagonal blocks1364, 1674
- swapping distributed vectors2829
- swapping vectors71
- Sylvester's equation986
- symmetric band matrix
 - 1-norm value1405
 - Frobenius norm1405
 - infinity-norm1405
 - largest absolute value of element1405
- symmetric distributed matrix
 - rank-n update2879, 2881
 - scalar-matrix-matrix product2877
- Symmetric Eigenproblems1063
- symmetric indefinite matrix
 - factorization with diagonal pivoting method1668, 1670
 - matrix-vector product1778
- symmetric matrix

- Bunch-Kaufman factorization
 - packed storage458
- eigenvalues and eigenvectors2034, 2036, 2039, 2043
 - estimating the condition number
 - packed storage524
- generalized eigenvalue problems929
 - inverting the matrix
 - packed storage619
 - matrix-vector product
 - band storage99
 - packed storage101, 1296
- packed storage463
 - rank-1 update
 - packed storage103, 1298
 - rank-2 update
 - packed storage105
- rank-2k update145
- rank-n update143
 - reducing to standard form
 - LAPACK1665
 - ScaLAPACK2262
 - reducing to tridiagonal form
 - LAPACK1612
 - ScaLAPACK2215
- scalar-matrix-matrix product140
- scaling2176
 - solving systems of linear equations
 - packed storage492
- symmetric matrix in packed form
 - 1-norm value1408
 - Frobenius norm1408
 - infinity- norm1408
 - largest absolute value of element1408
- symmetric positive definite distributed matrix
 - computing scaling factors1880
 - equilibration1880
- symmetric positive semidefinite matrix
 - Cholesky factorization438
- symmetric positive-definite band matrix
 - Cholesky factorization1655
- symmetric positive-definite distributed matrix
 - inverting the matrix1875
- symmetric positive-definite matrix
 - Cholesky factorization
 - band storage443, 1833
 - LAPACK1657
 - packed storage441
 - ScaLAPACK1832, 2257
 - estimating the condition number
 - band storage514
 - packed storage512
 - tridiagonal matrix515
 - inverting the matrix
 - packed storage604
 - solving systems of linear equations
 - band storage478, 1848
 - LAPACK472
 - packed storage476
 - ScaLAPACK1847
- symmetric positive-definite tridiagonal matrix
 - solving systems of linear equations1851
- symmetric tridiagonal matrix
 - eigenvalues and eigenvectors2290
- system of linear equations
 - with a distributed triangular matrix2861
 - with a triangular matrix
 - band storage114
 - packed storage119
- systems of linear equations
 - linear equations2329

- systems of linear equationslinear equations1838
- syswapr1660
- syswapr1663
- sytri2612
- sytri2x616

T

- Task Computation Routines3106
- Task Creation and Initialization3090
- Task Status3087
- threading control
 - mkl_domain_get_max_threads2986
 - mkl_domain_set_num_threads2981
 - mkl_get_dynamic2987
 - mkl_get_max_threads2985
 - mkl_set_dynamic2984
 - mkl_set_num_threads2981
 - mkl_set_num_threads_local2982
- Threading Control2978, 2979
- timing functions
 - mkl_get_clocks_frequency2994
 - MKL_Get_Cpu_Clocks2993
 - mkl_get_cpu_frequency2993
 - mkl_get_max_cpu_frequency2994
 - second/dsecnd2992
- TR routines
 - ?trnlspl_check2955
 - ?trnlspl_delete2959
 - ?trnlspl_get2958
 - ?trnlspl_init2953
 - ?trnlspl_solve2956
 - ?trnlsplbc_check2962
 - ?trnlsplbc_delete2967
 - ?trnlsplbc_get2966
 - ?trnlsplbc_init2960
 - ?trnlsplbc_solve2965
 - nonlinear least squares problem
 - with linear bound constraints2960
 - without constraints2952
 - organization and implementation2951
- transposition
 - distributed complex matrix2885
 - distributed complex matrix, conjugated2886
 - distributed real matrix2884
- Transposition and General Memory Movement Routines387
- transposition parameter3147
- trapezoidal matrix
 - 1-norm value1418
 - copying2340
 - Frobenius norm1418
 - infinity- norm1418
 - largest absolute value of element1418
 - reduction to triangular form2221
 - RZ factorization
 - LAPACK822
 - ScaLAPACK1933
- trexc980
- triangular band matrix
 - 1-norm value1415
 - Frobenius norm1415
 - infinity- norm1415
 - largest absolute value of element1415
- triangular banded equations
 - LAPACK1579
 - ScaLAPACK2250
- triangular distributed matrix
 - inverting the matrix1877
 - scalar-matrix-matrix product2888

- triangular factorization
 - band matrix430, 1825, 1827, 2106, 2327
 - diagonally dominant tridiagonal matrix
 - LAPACK434
 - general matrix425, 1823
 - Hermitian matrix
 - packed storage461
 - Hermitian positive semidefinite matrix438
 - Hermitian positive-definite matrix
 - band storage443, 1833
 - packed storage441
 - tridiagonal matrix444, 1836
 - symmetric matrix
 - packed storage458, 463
 - symmetric positive semidefinite matrix438
 - symmetric positive-definite matrix
 - band storage443, 1833
 - packed storage441
 - tridiagonal matrix444, 1836
 - tridiagonal matrix
 - LAPACK432
 - ScaLAPACK2328
- triangular matrix
 - 1-norm value
 - LAPACK1418
 - ScaLAPACK2153
 - copying1701, 1702, 1709, 1711–1713
 - estimating the condition number
 - band storage532
 - packed storage530
 - Frobenius norm
 - LAPACK1418
 - ScaLAPACK2153
 - infinity- norm
 - LAPACK1418
 - ScaLAPACK2153
 - inverting the matrix
 - LAPACK1680
 - packed storage625
 - ScaLAPACK2276
 - largest absolute value of element
 - LAPACK1418
 - ScaLAPACK2153
 - matrix-vector product
 - band storage112
 - packed storage117
 - product
 - blocked algorithm1621, 2225
 - LAPACK1620, 1621
 - ScaLAPACK2223, 2225
 - unblocked algorithm1620
 - ScaLAPACK1971
 - scalar-matrix-matrix product148
 - solving systems of linear equations
 - band storage114, 501
 - packed storage119, 499
 - ScaLAPACK1853
 - swapping adjacent diagonal blocks1674
 - triangular matrix factorization
 - Hermitian positive-definite matrix436
 - symmetric positive-definite matrix436
 - triangular matrix in packed form
 - 1-norm value1416
 - Frobenius norm1416
 - infinity- norm1416
 - largest absolute value of element1416
 - triangular pentagonal matrix
 - QR factorization836, 1704, 1706
 - triangular system of equations
 - solving with scale factor
 - LAPACK1614
 - ScaLAPACK2219
- tridiagonal matrix
 - base representations and eigenvalues2293, 2298
 - eigenvalues and eigenvectors2312, 2319
 - eigenvectors2304
 - estimating the condition number508
 - solving systems of linear equations
 - ScaLAPACK2329
 - tridiagonal system of equations1658
 - tridiagonal triangular factorization
 - band matrix2108
 - tridiagonal triangular system of equations2254
 - trigonometric transform
 - backward cosine2894
 - backward sine2894
 - backward staggered cosine2894
 - backward staggered sine2894
 - backward twice staggered cosine2894
 - backward twice staggered sine2894
 - forward cosine2894
 - forward sine2894
 - forward staggered cosine2894
 - forward staggered sine2894
 - forward twice staggered cosine2894
 - forward twice staggered sine2894
 - Trigonometric Transform interface
 - routines
 - ?_backward_trig_transform2902
 - ?_commit_trig_transform2898
 - ?_forward_trig_transform2900
 - ?_init_trig_transform2897
 - free_trig_transform2904
 - Trigonometric Transforms interface2897
 - TT interface2893
 - TT routines2897
 - two matrices
 - QR factorization
 - LAPACK830
 - ScaLAPACK1942

U

- ungbr855
- Uniform (continuous)2630
- Uniform (discrete)2657
- UniformBits2659
- UniformBits322660
- UniformBits642661
- unitary matrix
 - CS decomposition
 - LAPACK1034, 1039, 1184, 1188, 1622, 1625, 1628, 1631, 1635, 1637
 - from LQ factorization
 - LAPACK1642
 - ScaLAPACK2231
 - from QL factorization
 - LAPACK1640, 1645
 - ScaLAPACK2227, 2236
 - from QR factorization
 - LAPACK1641
 - ScaLAPACK2229
 - from RQ factorization
 - LAPACK1644
 - ScaLAPACK2234
- ScaLAPACK1971, 1986
- Unpack Functions2485
- updating
 - rank-1

distributed general matrix2836	v?Conj2497
distributed Hermitian matrix2846	v?Cos2528
distributed symmetric matrix2853	v?Cosh2544
general matrix81	v?Div2504
Hermitian matrix90, 95	v?Erf2557
real symmetric matrix103, 109	v?Erfc2559
rank-1, conjugated	v?ErfcInv2565
distributed general matrix2838	v?ErfInv2563
general matrix82	v?Exp2520
rank-1, unconjugated	v?ExpM12522
distributed general matrix2840	v?Floor2571
general matrix84	v?Frac2579
rank-2	v?Hypot2518
distributed Hermitian matrix2847	v?Inv2503
distributed symmetric matrix2855	v?InvCbrt2510
Hermitian matrix92, 97	v?InvSqrt2508
symmetric matrix105, 110	v?Lgamma2568
rank-2k	v?LGamma2568
Hermitian distributed matrix2874	v?LinearFrac2501
Hermitian matrix137	v?Ln2523
symmetric distributed matrix2881	v?Log102525
symmetric matrix145	v?Log1p2527
rank-k	v?Modf2578
distributed Hermitian matrix2872	v?Mul2494
Hermitian matrix135	v?MulByConj2496
symmetric distributed matrix2879	v?NearbyInt2576
rank-n	v?Pack2581
symmetric matrix143	v?Pow2513
updating:rank-1	v?Pow2o32511
Hermitian matrix	v?Pow3o22512
packed storage95	v?Powx2516
real symmetric matrix	v?Rint2577
packed storage103	v?Round2574
updating:rank-2	v?Sin2530
Hermitian matrix	v?SinCos2532
packed storage97	v?Sinh2546
symmetric matrix	v?Sqr2493
packed storage105	v?Sqrt2506
upper Hessenberg matrix	v?Sub2492
1-norm value	v?Tan2535
LAPACK1404	v?Tanh2549
ScaLAPACK2149	v?tgamma2570
Frobenius norm	v?TGamma2570
LAPACK1404	v?Trunc2573
ScaLAPACK2149	v?Unpack2583
infinity- norm	vcAdd2490
LAPACK1404	vcPackI2581
ScaLAPACK2149	vcPackM2581
largest absolute value of element	vcPackV2581
LAPACK1404	vcSin2530
ScaLAPACK2149	vcSub2492
ScaLAPACK1971	vcUnpackI2583
	vcUnpackM2583
	vcUnpackV2583
	vdAdd2490
	vdLgamma2568
	vdLGamma2568
	vdPackI2581
	vdPackM2581
	vdPackV2581
	vdSin2530
	vdSub2492
	vdTgamma2570
	vdTGamma2570
	vdUnpackI2583
	vdUnpackM2583
	vdUnpackV2583
	vector arguments
	array dimension3143
	default3144

V

v?Abs2498
v?Acos2537
v?Acosh2551
v?Add2490
v?Arg2500
v?Asin2539
v?Asinh2553
v?Atan2541
v?Atan22542
v?Atanh2555
v?Cbrt2509
v?CdfNorm2561
v?CdfNormInv2567
v?Ceil2572
v?CIS2534

- examples3143
- increment3143
- length3143
- matrix one-dimensional substructures3143
- sparse vector154
- vector conjugation1291, 2100
- vector indexing2486
- vector mathematical functions
 - absolute value2498
 - addition2490
 - argument2500
 - complementary error function value2559
 - complex exponent of real vector elements2534
 - computing a rounded integer value and raising
 - inexact result exception2577
 - computing a rounded integer value in current
 - rounding mode2576
 - computing a truncated integer value2578
 - conjugation2497
 - cosine2528
 - cube root2509
 - cumulative normal distribution function value2561
 - denary logarithm2525
 - division2504
 - error function value2557
 - exponential2520
 - exponential of elements decreased by 12522
 - four-quadrant arctangent2542
 - gamma function2568, 2570
 - hyperbolic cosine2544
 - hyperbolic sine2546
 - hyperbolic tangent2549
 - inverse complementary error function value2565
 - inverse cosine2537
 - inverse cube root2510
 - inverse cumulative normal distribution function
 - value2567
 - inverse error function value2563
 - inverse hyperbolic cosine2551
 - inverse hyperbolic sine2553
 - inverse hyperbolic tangent2555
 - inverse sine2539
 - inverse square root2508
 - inverse tangent2541
 - inversion2503
 - linear fraction transformation2501
 - multiplication2494
 - multiplication of conjugated vector element2496
 - natural logarithm2523
 - natural logarithm of vector elements increased by
 - 12527
 - power2513
 - power (constant)2516
 - power 2/32511
 - power 3/22512
 - rounding to nearest integer value2574
 - rounding towards minus infinity2571, 2579
 - rounding towards plus infinity2572
 - rounding towards zero2573
 - scaling1777
 - scaling, reciprocal1776
 - sine2530
 - sine and cosine2532
 - square root2506
 - square root of sum of squares2518
 - squaring2493
 - subtraction2492
 - tangent2535
- Vector Mathematical Functions2483
- vector multiplication
 - LAPACK1659
 - ScaLAPACK2261
 - vector pack function2581
 - vector statistics functions
 - Bernoulli2663
 - Beta2654
 - Binomial2666
 - Cauchy2643
 - CopyStream2615
 - CopyStreamState2616
 - DeleteStream2614
 - dNewAbstractStream2611
 - Exponential2637
 - Gamma2652
 - Gaussian2632
 - GaussianMV2634
 - Geometric2664
 - GetBrngProperties2676
 - GetNumRegBrngs2625
 - GetStreamSize2621
 - GetStreamStateBrng2625
 - Gumbel2650
 - Hypergeometric2668
 - iNewAbstractStream2609
 - Laplace2639
 - LeapfrogStream2621
 - LoadStreamF2617
 - LoadStreamM2620
 - Lognormal2648
 - NegBinomial2673
 - NewStream2607
 - NewStreamEx2608
 - Poisson2669
 - PoissonV2671
 - Rayleigh2645
 - SaveStreamF2616
 - SaveStreamM2619
 - SkipAheadStream2623
 - sNewAbstractStream2613
 - Uniform (continuous)2630
 - Uniform (discrete)2657
 - UniformBits2659
 - UniformBits322660
 - UniformBits642661
 - Weibull2641
 - vector unpack function2583
 - vector-scalar product
 - sparse vectors155
 - vectors
 - adding magnitudes of vector elements54
 - copying57
 - dot product
 - complex vectors61
 - complex vectors, conjugated60
 - real vectors58
 - element with the largest absolute value72
 - element with the largest absolute value of real part
 - and its index2104
 - element with the smallest absolute value73
 - Euclidean norm63
 - Givens rotation65
 - linear combination of vectors56, 388
 - modified Givens transformation parameters68
 - rotation of points64
 - rotation of points in the modified plane66
 - sparse vectors154
 - sum of vectors56, 388
 - swapping71
 - vector-scalar product69
 - viRngUniformBits2659

viRngUniformBits322660	ClearErrorCallBack2591
viRngUniformBits642661	clearerrorstatus2588
VM	GetErrorCallBack2591
Functions Interface2485	geterrstatus2588
Input Parameters2486	getmode2586
Output Parameters2486	SetErrorCallBack2589
VM arithmetic functions2490	SetErrStatus2587
VM exponential and logarithmic functions2519	SetMode2585
VM functions	VM hyperbolic functions2544
mathematical functions	VM mathematical functions
v?Abs2498	arithmetic2490
v?Acos2537	exponential and logarithmic2519
v?Acosh2551	hyperbolic2544
v?Add2490	power and root2503
v?Arg2500	rounding2571
v?Asin2539	special2557
v?Asinh2553	special value notations2489
v?Atan2541	trigonometric2528
v?Atan22542	VM Mathematical Functions2485
v?Atanh2555	VM Pack Functions2485
v?Cbrt2509	VM Pack/Unpack Functions2580
v?CdfNorm2561	VM power and root functions2503
v?CdfNormInv2567	VM rounding functions2571
v?Ceil2572	VM Service Functions2584
v?CIS2534	VM special functions2557
v?Conj2497	VM trigonometric functions2528
v?Cos2528	vmcAdd2490
v?Cosh2544	vmcSin2530
v?Div2504	vmcSub2492
v?Erf2557	vmdAdd2490
v?Erfc2559	vmdSin2530
v?ErfcInv2565	vmdSub2492
v?ErfInv2563	vmlClearErrorCallBack2591
v?Exp2520	vmlclearerrstatus2588
v?Exp12522	vmlGetErrorCallBack2591
v?Floor2571	vmlgeterrstatus2588
v?Frac2579	vmlgetmode2586
v?Hypot2518	vmlSetErrorCallBack2589
v?Inv2503	vmlSetErrorStatus2587
v?InvCbrt2510	vmlSetMode2585
v?InvSqrt2508	vmsAdd2490
v?LGamma2568	vmsSin2530
v?LinearFrac2501	vmsSub2492
v?Ln2523	vmzAdd2490
v?Log102525	vmzSin2530
v?Log1p2527	vmzSub2492
v?Modf2578	VS Fortran header2593
v?Mul2494	VS routines
v?MulByConj2496	advanced service routines
v?NearbyInt2576	GetBrngProperties2676
v?Pow2513	convolution/correlation
v?Pow2o32511	CopyTask2705
v?Pow3o22512	DeleteTask2704
v?Powx2516	Exec2695
v?Rint2577	Exec1D2697
v?Round2574	ExecX2699
v?Sin2530	ExecX1D2702
v?SinCos2532	NewTask2682
v?Sinh2546	NewTask1D2684
v?Sqr2493	NewTaskX2685
v?Sqrt2506	NewTaskX1D2688
v?Sub2492	SetInternalPrecision2692
v?Tan2535	generator routines
v?Tanh2549	Bernoulli2663
v?TGamma2570	Beta2654
v?Trunc2573	Binomial2666
pack/unpack functions	Cauchy2643
v?Pack2581	Exponential2637
v?Unpack2583	Gamma2652
service functions	Gaussian2632

GaussianMV2634
 Geometric2664
 Gumbel2650
 Hypergeometric2668
 Laplace2639
 Lognormal2648
 NegBinomial2673
 Poisson2669
 PoissonV2671
 Rayleigh2645
 Uniform (continuous)2630
 Uniform (discrete)2657
 UniformBits2659
 UniformBits322660
 UniformBits642661
 Weibull2641
 service routines
 CopyStream2615
 CopyStreamState2616
 DeleteStream2614
 dNewAbstractStream2611
 GetNumRegBrngs2625
 GetStreamSize2621
 GetStreamStateBrng2625
 iNewAbstractStream2609
 LeapfrogStream2621
 LoadStreamF2617
 LoadStreamM2620
 NewStream2607
 NewStreamEx2608
 SaveStreamF2616
 SaveStreamM2619
 SkipAheadStream2623
 sNewAbstractStream2613
 summary statistics
 Compute2750
 DeleteTask2751
 EditCorParameterization2746
 EditCovCor2732
 EditCP2734
 EditMissingValues2743
 EditMoments2730
 EditOutliersDetection2741
 EditPooledCovariance2738
 EditQuantiles2736
 EditRobustCovariance2740
 EditStreamQuantiles2737
 EditSums2731
 EditTask2721
 NewTask2718
 VS routines: convolution/correlation
 SetInternalDecimation2694
 SetMode2691
 SetStart2693
 VS task2593
 vsAdd2490
 vslConvCopyTask2705
 vslCorrCopyTask2705
 vsldsscompute2750
 vsldSSCompute2750
 vsldsseditcorparameterization2746
 vsldSSEditCorParameterization2746
 vsldsseditcovcor2732
 vsldSSEditCovCor2732
 vsldsseditcp2734
 vsldSSEditCP2734
 vsldsseditmissingvalues2743
 vsldSSEditMissingValues2743
 vsldsseditmoments2730
 vsldSSEditMoments2730
 vsldsseditoutliersdetection2741
 vsldSSEditOutliersDetection2741
 vsldsseditpartialcovcor2735
 vsldsseditpooledcovariance2738
 vsldSSEditPooledCovariance2738
 vsldsseditquantiles2736
 vsldSSEditQuantiles2736
 vsldsseditrobustcovariance2740
 vsldSSEditRobustCovariance2740
 vsldsseditstreamquantiles2737
 vsldSSEditStreamQuantiles2737
 vsldsseditsums2731
 vsldSSEditSums2731
 vsldssedittask2721
 vsldSSEditTask2721
 vsldssnewtask2718
 vsldSSNewTask2718
 vsPackI2581
 vsPackM2581
 vsPackV2581
 vsSin2530
 vsSub2492
 vstgamma2570
 vsTGamma2570
 vsUnpackI2583
 vsUnpackM2583
 vsUnpackV2583
 vzAdd2490
 vzPackI2581
 vzPackM2581
 vzPackV2581
 vzSin2530
 vzSub2492

vzUnpackI2583
 vzUnpackM2583
 vzUnpackV2583

W

Weibull2641
 Wilkinson transform2226

X

xerbla2988
 xerbla_array1812
 xerbla, error reporting routine51, 2487, 2815

Z

zbbcsd1034
 zdla_gercond_c1737
 zdla_gercond_x1739
 zgbcon506
 zgbrfsx545
 zgbsvx666
 zgbtrs467
 zgecon504
 zgemmt394
 zgeqpf776
 zgtrfs552
 zhegs21665
 zheswapr1662
 zhetd21666
 zhetri2614
 zhetri2x618
 zhetrs2491
 zhgeqz996
 zhseqr962
 zla_gbamv1719
 zla_gbrcond_c1723
 zla_gbrcond_x1725
 zla_gbrfsx_extended1726
 zla_gbrpvgrw1733
 zla_geamv1734
 zla_gerfsx_extended1740
 zla_gerpvgrw1775
 zla_heamv1746
 zla_hercond_c1748
 zla_hercond_x1749
 zla_herfsx_extended1751
 zla_herpvgrw1757
 zla_lin_berr1758
 zla_porcond_c1760
 zla_porcond_x1762
 zla_porfsx_extended1763
 zla_porpvgrw1770
 zla_syamv1778
 zla_syrcond_c1781
 zla_syrcond_x1783
 zla_syrfsx_extended1784
 zla_syrpvgrw1791
 zla_wwaddw1792
 zlag2c1684

zlakf21386
 zlapmr1422
 zlapmt1423
 zlarfb1464
 zlarft1471
 zlarge1475
 zlarnd1477
 zlaror1479
 zlarot1482
 zlarocl21776
 zlascl21777
 zlat2c1717
 zlatm11581
 zlatm21583
 zlatm31586
 zlatm51589
 zlatm61593
 zlatme1596
 zlatmr1600
 zlatps1609
 zlatrd1612
 zlatrs1614
 zlatrz1618
 zlauu21620
 zlauum1621
 zpbtfd21655
 zporfsx557
 zpotf21657
 zpprfs563
 zpptrs476
 zppts21658
 zrsc1659
 zsysconv1299
 zsyswapr1660
 zsyswapr11663
 zsytf21668
 zsytf2_rook1670
 zsytri2612
 zsytri2x616
 zsytrs2489
 ztgex21674
 ztgsy21677
 ztrexc980
 ztrti21680
 zunbdb1039
 zunbdb11622
 zunbdb21625
 zunbdb31628
 zunbdb41631
 zunbdb51635
 zunbdb61637
 zuncsd1184
 zuncsd2by11188
 zung2l1640
 zung2r1641
 zungbr855
 zungl21642
 zungr21644
 zunm2l1645
 zunm2r1647
 zunml21649
 zunmr21651
 zunmr31653